A basis for intrusion detection in distributed systems using kernel-level data tainting.

Christophe Hauser

N° d'ordre : 2013-13-TH ANNÉE 2013

THÈSE DE DOCTORAT

Mention : Informatique

École doctorale MATISSE

présentée par

Christophe Hauser

préparée à l'unité de recherche CIDre
Confidentialité Intégrité Disponibilité Réparti
Supélec/INRIA

Détection d'intrusions dans les systèmes distribués par propagation de teinte au niveau noyau

—

A basis for intrusion detection in distributed systems using kernel-level data tainting

Thèse soutenue à Rennes le 19 juin 2013

devant le jury composé de :

Hervé Debar
Professeur à Télécom SudParis / rapporteur

Alexander Pretschner
Professeur à Technische Universität München / rapporteur

Anne-Marie Kermarrec
Directrice de recherche à l'INRIA/ présidente

Ernest Foo
Professeur à Queensland University of Technology / examinateur

Ludovic Mé
Professeur à Supélec / directeur de thèse en France

Frédéric Tronel
Professeur associé à Supélec / co-directeur de thèse

Colin Fidge
Professeur à Queensland University of Technology/ Directeur de thèse en Australie

Résumé en Français

Les systèmes d'information actuels, qu'il s'agisse de réseaux d'entreprises, de services en ligne ou encore d'organisations gouvernementales, reposent très souvent sur des systèmes distribués, impliquant un ensemble de machines fournissant des services internes ou externes. La sécurité de tels systèmes d'information est construite à plusieurs niveaux (défense en profondeur). Lors de l'établissement de tels systèmes, des politiques de contrôle d'accès, d'authentification, de filtrage (firewalls, *etc.*) sont mises en place afin de garantir la sécurité des informations. Cependant, ces systèmes sont très souvent complexes, et évoluent en permanence. Il devient alors difficile de maintenir une politique de sécurité sans faille sur l'ensemble du système (quand bien même cela serait possible), et de résister aux attaques auxquelles ces services sont quotidiennement exposés. C'est ainsi que les *systèmes de détection d'intrusions* sont devenus nécessaires, et font partie du jeu d'outils de sécurité indispensables à tous les administrateurs de systèmes exposés en permanence à des attaques potentielles.

Les systèmes de détection d'intrusions se classifient en deux grandes familles, qui diffèrent par leur méthode d'analyse: l'approche par scénarios et l'approche comportementale. L'approche par scénarios est la plus courante, et elle est utilisée par des systèmes de détection d'intrusions bien connus tels que Snort [59], Prélude [75] et d'autres. Cette approche consiste à reconnaître des signatures d'attaques connues dans le trafic réseau (pour les IDS réseau) et des séquences d'appels systèmes (pour les IDS hôtes). Il s'agit donc de détecter des comportements anormaux du système liés à la présence d'attaques. Bien que l'on puisse ainsi détecter un grand nombre d'attaques, cette approche ne permet pas de détecter de nouvelles attaques, pour lesquelles aucune signature n'est connue. Par ailleurs, les malwares modernes emploient souvent des techniques dites de *morphisme binaire*, afin d'échapper à la détection par signatures. L'approche comportementale, à l'inverse de l'approche par signature, se base sur la modélisation du fonctionnement normal du système. Cette approche permet ainsi de détecter de nouvelles attaques tout comme des attaques plus anciennes, n'ayant recours à aucune base de données de connaissance d'attaques existantes. Il existe plusieurs types d'approches comportementales, certains modèles sont statistiques, d'autres modèles s'appuient sur une politique de sécurité.

Dans cette thèse, on s'intéresse à la détection d'intrusions dans des systèmes distribués, en adoptant une approche comportementale basée sur une politique de sécurité. Elle est exprimée sous la forme d'une politique de flux d'information. Les flux d'informations sont suivis via une technique de propagation de marques (appelée en anglais *taint marking*) appliquées sur les objets du système d'exploitation, directement au niveau du noyau. De telles approches existent également au niveau langage (par exemple par instrumentation de la machine virtuelle Java, ou bien en modifiant le code des applications) [50, 51] ou encore au niveau de l'architecture [67, 78] (en émulant le microprocesseur afin de tracer les flux d'information entre les registres, pages mémoire *etc.*), et permettent ainsi une analyse fine des flux d'informations. Cependant, nous avons choisi de nous placer au niveau du système d'exploitation, afin de satisfaire les objectifs suivants:

- Détecter les intrusions à tous les niveaux du système, pas spécifiquement au sein d'une ou plusieurs applications.
- Déployer notre système en présence d'applications *natives*, dont le code source n'est pas nécessairement disponible (ce qui rend leur instrumentation très difficile voire impossible).
- Utiliser du matériel standard présent sur le marché. Il est très difficile de modifier physiquement les microprocesseurs, et leur émulation a un impact très important sur les performances

du système.

Vue d'ensemble

Nous avons ainsi étendu un modèle de propagation de marques, en nous appuyant sur des techniques existantes, issues de précédents travaux au sein de l'équipe CIDre. Ensuite, ce modèle de propagation a été implémenté via la réalisation d'un prototype. Ce nouveau modèle permet de prendre en compte les spécificités du suivi de flux d'information dans un système d'exploitation de type Unix, mais peut aussi être utilisé dans des environnements distribués. Ce modèle attache des marques (ou *tags*) aux objets du système d'exploitation, dans le but de suivre leur propagation tout au long de la vie du système. Les objets tels que les fichiers, les processus et les sockets réseau sont ainsi marqués par chaque flux d'information. Nous avons implémenté ce modèle dans le noyau Linux, en tant que module de sécurité. La conception et l'implémentation de ce modèle représentent la **première contribution** de cette thèse. Nous avons publié et présenté ce modèle lors de la conférence internationale ICC 2011 (IEEE International Conference on Communications) [65].

Nous avons ensuite étendu ces travaux afin de prendre en considération les flux d'information sur le réseau. Cette extension du modèle permet de définir une politique réseau afin de contrôler les interactions autorisées entre les applications ou utilisateurs vis à vis de l'information surveillée. Cette politique définit d'une part quelles informations sont autorisées à quitter le système via le réseau, et d'autre part dans quelles conditions de nouvelles informations, arrivant par le réseau depuis des sources connues ou inconnues, sont autorisées à se mélanger avec des informations existantes sur le système surveillé. Cette politique est définie de manière globale au système. Les règles qui concernent l'information sortante protègent la confidentialité des données, tandis que les règles qui concernent l'information entrante protègent leur intégrité. La possibilité de définir une telle politique pour protéger des données privées offre de nouvelles solutions quant à la détection de violations de la vie privée ou au vol d'informations personnelles. Cette **seconde contribution** a été publiée et présentée lors de la conférence internationale AISC 2012 (Australasian Information Security Conference) [32].

Enfin, notre dernière contribution concerne la généralisation du précédent modèle à la détection d'intrusions en environnement distribué. En prenant de multiples machines (que nous réunissons en groupes de machines) en considération, il devient possible de définir une politique adaptée à des systèmes plus complexes, tout en gardant une approche "à grain fin", c'est à dire en conservant une spécification fine de la politique. Une telle politique est définie à l'échelle d'un groupe de machines. Elle est distribuée au sein de chaque machine du groupe, et définit les interactions autorisées entre processus de machines différentes, ainsi qu'entre processus locaux. Cette **dernière contribution** a donné lieu à une publication, qui a été acceptée et présentée lors de la conférence internationale ICC 2013 [31].

Modèle de détection

Notre modèle de détection fait intervenir des marques appelées *tags* afin de suivre les flux d'information entre objets du système d'exploitation surveillé. Ces objets sont considérés comme conteneurs d'information, et à tout moment, nous souhaitons pouvoir déterminer le contenu de chaque objet afin de vérifier qu'il correspond à un état normal du système. La spécification de cet état normal, ou contenu normal, se fait via une politique de sécurité. Cette politique dissocie les données passives du code actif des applications: le code d'une application est considéré comme passif lorsqu'il est stocké dans un fichier, mais il est considéré comme actif lorsque qu'il est en cours d'exécution. Cette distinction nous permet d'exprimer finement la politique de sécurité. Nous considérons ainsi comme étant de *l'information* tout élément passif (donnée ou code stockés) ou actif (code en cours d'exécution).

La définition de la politique ainsi que le suivi de flux d'informations font intervenir quatre types de *tags*:

- Les tags d'information, ou *information tags*, décrivent le contenu des objets (ou *conteneurs*) auxquels ils sont attachés, à tout instant. Ils contiennent des meta-informations, permettant

de décrire individuellement chaque élément d'information.

- Les tags de politique, ou *policy tags*, décrivent la politique des objets auxquels ils sont attachés. Ils décrivent quelles sont les combinaisons légales d'information que ces objets peuvent contenir. Toute déviation vis à vis de cette politique indique un comportement anormal du système.
- Les tags de politique d'exécution, ou *execute policy tags*, décrivent le comportement légal des processus résultant de l'exécution de code marqué. Ils sont attachés aux fichiers exécutables. Ces tags ne sont utilisés qu'au moment de l'exécution, afin de déterminer les tags de politique des processus.
- Le tag de politique réseau, ou *network policy tag*, détermine les interactions légales entre processus et données vis à vis du réseau. Il détermine quels processus (en se basant sur la marque du code exécuté) peuvent légalement recevoir ou envoyer quelles informations à quels autres processus distants, au sein d'un système distribué. Il n'existe qu'un seul tag de politique réseau par machine, celui-ci définit toutes les interactions légales entre processus, information et réseau.

Ce modèle a été implémenté dans le noyau Linux, sous la forme d'un module de sécurité. Nous suivons les flux d'information entre les fichiers, les sockets réseau, les zones de mémoire partagée, les files de messages, les inodes *etc.* Cette implémentation utilise des mécanismes standard du noyau, et les opérations complexes utilisent des structures de données optimisées afin de limiter l'impact sur les performances. Le code a été testé sur plusieurs architectures, et a été reporté comme fonctionnel sur la plateforme Android.

Résultats expérimentaux

Les travaux réalisés au sein de cette thèse ont été vérifiés expérimentalement dans plusieurs cas de figure. Outre les tests de validation de l'implémentation vis à vis du modèle mis en œuvre, nous avons réalisé deux scénarios correspondant à des cas d'utilisation réels d'un système de détection d'intrusions. Dans le premier cas, les attaques contre la confidentialité ont été visées. Ce premier scénario met en œuvre une attaque contre le navigateur web Firefox[1], en utilisant une version vulnérable du plugin Java[2], et vise à valider notre approche quant à la détection de fuites d'informations impliquant des données confidentielles, via l'exploitation d'une vulnérabilité (CVE 2008-5353) au sein d'une page web malveillante. Afin de détecter de telles fuites d'informations, nous avons tout d'abord marqué chaque information confidentielle avec un *tag* unique, puis nous avons configuré le système avec une politique de sécurité interdisant l'émission d'informations marquées. La propagation de marques entre les objets du système permet ainsi de suivre les informations de bout en bout, et de lever une alerte lorsque des informations marquées arrivent au niveau des sockets réseau.

Le second scénario mis en ouvre s'applique aux systèmes distribués. Nous avons considéré un ensemble de plusieurs machines supervisées, fournissant un service web distribué, composé d'un serveur web (Apache), d'un serveur de bases de données (PostgreSQL) et du moteur de blog Wordpress[3]. Le moteur de blog utilise le plugin de e-commerce *Foxypress*, qui présente une vulnérabilité (EDB-ID 18991). Cette vulnérabilité permet l'upload de fichiers arbitraires et l'exécution de code à distance sur la machine qui héberge le service vulnérable. Le serveur web et la base de données hébergent deux sites web, l'un étant public et accessible depuis l'internet, et l'autre privé et accessible uniquement depuis le réseau local. Notre objectif ici était de démontrer la capacité de notre système de détection d'intrusion à détecter les attaques réussies, non seulement au niveau de la machine directement visée, mais également au niveau de chaque machine qui compose le système distribué, afin de pouvoir émettre un diagnostic plus riche de l'attaque a-posteriori. L'attaque que nous avons mis en œuvre implique un attaquant extérieur qui souhaite accéder aux informations confidentielles du site web privé.

[1] `http://www.mozilla.org/firefox/`
[2] http://www.oracle.com/technetwork/java/
[3] http://www.wordpress.org

Nous avons ainsi déployé une politique de sécurité décrivant le comportement légal des processus composant le service distribué, localement sur chaque système ainsi que sur le réseau lors de leurs communications. Cette politique autorise le serveur web ainsi que le serveur de bases de données à traiter des requêtes concernant *un seul* des deux sites web à la fois. Ceci est rendu possible par le fait que, Apache et PostgreSQL créent un nouveau processus pour traiter chaque connexion, et en aucun cas les informations des deux sites ne sont mélangées lorsque le système fonctionne normalement. L'attaque que nous avons déployée injecte un script PHP contenant du code malveillant, en utilisant la vulnérabilité présentée précédemment, sur le site public (seul site accessible depuis l'extérieur). L'attaquant a ainsi la main sur le processus en question, et peut désormais effectuer des requêtes concernant le second site web. Dès lors qu'il effectue une telle requête, le processus attaqué, qui jusqu'alors était marqué avec des informations du site web public, se voit également marqué avec des informations du site web privé, et viole ainsi la politique de sécurité. Une alerte est levée sur la machine locale (le serveur web), et toute connexion entre le processus infecté et un processus d'une autre machine supervisée provoque la contamination de ce dernier, levant ainsi des alertes sur les autres machines.

Évaluation

Une évaluation de notre modèle et de son implémentation est présentée en conclusion du chapitre 8. En terme de performances, notre implémentation ajoute une pénalité maximale de 30% en terme de consommation mémoire, et de 40% en terme de consommation CPU. Le temps maximal d'exécution de certaines opérations peut également s'élever à 300% dans des conditions extrêmes, limite dûe à une utilisation excessive du système de fichier, que l'on estime aisément contournable à l'aide d'optimisations (présentées dans la section 7.8).

L'évaluation de systèmes de détection d'intrusions fait généralement intervenir la notion de taux de *faux positifs* et de *faux négatifs*. Par conception, notre approche est conservatrice et surapproxime à tout moment la quantité d'information impliquée dans les flux d'informations. Ceci a pour effet de limiter très fortement la présence de *faux négatif*, qui à l'exception de canaux cachés ou de défauts dans la définition de la politique de sécurité, sont considérés comme inexistants dans notre système. Par ailleurs, le taux de *faux positifs* est directement lié à la précision avec laquelle nous observons les flux d'information. Nous identifions ainsi deux cas de figure: les cas où nous sommes contraints d'effectuer une forte surapproximation, par exemple lors de l'utilisation de mémoire partagée entre plusieurs processus, et les cas où nous effectuons une surapproximation plus modérée. Dans le premier cas, un grand nombre de *faux positifs* est généré, rendant difficile l'utilisation de notre système. Ceci est dû au niveau d'abstraction auquel nous nous plaçons dans le système. Depuis le noyau, il est impossible d'observer de façon exacte les accès à la mémoire effectués par les applications. Il s'agit de la principale limitation de notre approche, et nous envisageons plusieurs solutions afin d'affiner l'analyse des flux. Dans le second cas, la précision de notre analyse est plus fine, et nous sommes ainsi capables de détecter les intrusions avec un faible taux de *faux positifs*. Ces aspects sont présentés plus en détails dans la section 8.4 de ce manuscrit.

Nous avons ainsi mis en œuvre et implémenté un modèle de détection d'intrusions au niveau noyau, capable de détecter les intrusions aussi bien dans des machines isolées, qu'au sein de systèmes distribués. La mise en œuvre d'expérimentations nous a permis de valider notre approche de détection, et d'identifier ses limitations. Des travaux en cours au sein de l'équipe CIDre s'appuient sur notre travail, et ont pour objectif de mettre en œuvre des mécanismes de coopération entre des moniteurs de suivi de flux à plusieurs niveaux (niveau langage et niveau système d'exploitation), visant ainsi un affinement du suivi de flux afin de réduire les taux de *faux positifs*.

Cette thèse est organisée de la manière suivante: la première partie, composée des deux premiers chapitres, présente le contexte de recherche dans lequel notre travail s'inscrit. Le chapitre 1 introduit les fondements de notre approche, ainsi que les travaux précédents existants dans la littérature. Le chapitre 2 compare la base de notre modèle avec les modèles classiques de contrôle d'accès et de contrôle de flux d'information.

La seconde partie de cette thèse présente notre première contribution. Le chapitre 3 détaille notre modèle de détection d'intrusions, et le chapitre 4 présente son implémentation.

Enfin, la dernière partie de cette thèse présente l'extension de notre modèle au réseau et aux sys-

tèmes distribués, dans les chapitres 5 et 6, suivie de nos résultats expérimentaux dans le chapitre 8.

Abstract

Modern organisations rely intensively on information and communication technology infrastructures. Such infrastructures offer a range of services from simple mail transport agents or blogs to complex e-commerce platforms, banking systems or service hosting, and all of these depend on distributed systems. The security of these systems, with their increasing complexity, is a challenge. Cloud services are replacing traditional infrastructures by providing lower cost alternatives for storage and computational power, but at the risk of relying on third party companies. This risk becomes particularly critical when such services are used to host privileged company information and applications, or customers' private information. Even in the case where companies host their own information and applications, the advent of BYOD (Bring Your Own Device [48]) leads to new security related issues.

In response, our research investigated the characterization and detection of malicious activities at the operating system level and in distributed systems composed of multiple hosts and services. We have shown that intrusions in an operating system spawn abnormal information flows, and we developed a model of dynamic information flow tracking, based on *taint marking* techniques, in order to detect such abnormal behavior. We track information flows between objects of the operating system (such as files, sockets, shared memory, processes, *etc.*) and network packets flowing between hosts. This approach follows the anomaly detection paradigm. We specify the legal behavior of the system with respect to an information flow policy, by stating how users and programs from groups of hosts are allowed to access or alter each other's information. Illegal information flows are considered as intrusion symptoms. We have implemented this model in the Linux kernel[4], as a Linux Security Module (LSM), and we used it as the basis for practical demonstrations. The experimental results validated the feasibility of our new intrusion detection principles.

This research is part of a joint research project between Supélec (École supérieure d'éléctricité) and QUT (Queensland University of Technology).

[4]The source code is available at `http://www.blare-ids.org`.

Acknowledgements

The completion of this thesis marks the end of an amazing period, in which I met great people and had the opportunity to work on so many interesting things. Spending a year at QUT gave me the ability to discover a new culture, and to gather different points of view and approaches on the research side. This was a most exciting experience.

First of all, I would like to express my gratitude to my advisors, Frédéric Tronel and Ludovic Mé for their encouragements, criticisms and support. Their tremendous knowledge of the field and all their advices have been very helpful to complete this thesis.

I am also very grateful to Colin Fidge, Jason Reid and Andrew Clark for their welcoming and their support during my stay in Australia. Their guidance and their broad skills were very appreciable. In particular, many thanks to Andrew Clark, without whom this cooperation between Supélec and QUT would probably not have been possible.

I would like to thank all the members of the CIDre team for all the interesting discussions we had during the last few years, and all the members of the Information Security Institute for welcoming me in Australia.

Special thanks to Sajal, Gleb, Mr Kush, Ken and Mark for all the enjoyable moments we spent together.

Finally, thanks to my family and Sarah for supporting me throughout this long road.

Contents

List of Figures

Introduction

Over the last decade, the huge development of internet and home networks led to new online services, social networks and online mass market. Information systems have been expanded to fit more and more users with increasing data volumes. This made distributed systems very common and widely used. Nowadays, popular services store large amounts of user data online, "in the cloud". It is thus desirable that the underlying systems offer good security properties. Such security properties have to be defined and implanted into each system component through a *security policy*. This is defined as a set of rules specifying how the system is authorized to manage information, *i.e.,* what is legal within the system in terms of information and operations. Existing mechanisms have been designed to implant such policies, such as access control and firewalls. However, these are very difficult to maintain in complex growing environments, where perpetual bug fixes in software development make evasion possible for potential attackers.

As a result of this, intrusion detection systems (IDSes) have become a necessary addition to the security infrastructure of nearly every organization. IDSes typically record information from observed events and notify the system administrators when possibly illegal events occur. Most of the current approaches focus on *misuse detection*, by detecting patterns of abnormal behavior of the monitored system, *i.e.,* these are based on learned profiles or signatures of known attacks. Such approaches generally generate a high number of false positives, making it difficult for system administrators to successfully identify real attacks. Furthermore, these systems are not able to detect previously unseen attacks also known as "zero day attacks". An alternative approach to *misuse detection* is *anomaly detection*, describing deviations from an established normal state of the monitored system.

The aim of this research is to investigate the characterization and detection of malicious activities at the operating system level and in distributed systems composed of *groups* of hosts. Our approach follows the *anomaly detection* paradigm. It is based on a security policy describing the legal behavior of the system, an approach also known as *policy-based* intrusion detection. Detection of illegal activity is done by tracking information flows within the operating system and between hosts. An information flow policy defines the legal behavior of the system, by determining where information is allowed to flow, and which users or programs are allowed to access it. Any violation of this policy is considered as a symptom of intrusion, and raises an alert.

In order to achieve these goals, we have first designed and implemented a model of *taint marking*, labeling objects of the operating system with *tags*, so as to track their content by propagating taint data. Objects such as files, sockets and processes, amongst others, are tainted. It was implemented in the Linux kernel as a Linux security module. The design and implementation of this model represents our first contribution.

The consideration of network aspects, such as the policy regarding network interaction of applications, users and containers of information (*e.g.,* files, memory pages, *etc.*), represents our second contribution. This includes an extension of our model and implementation so as to take network sockets and packets into consideration. We introduced a *network policy*, defining the legality of information flows involving outgoing data, in terms of confidentiality, and incoming new data, in terms of integrity. It defines how new and possibly untrusted data is allowed to mix with data already present in the system. Specifying such a policy for *e.g.,* private user data offers a novel solution for tracking privacy violations caused by applications.

Finally, our last contribution is the generalization of this approach in order to detect intrusions in distributed systems. Taking multiple hosts into account (we gather hosts in *groups*, in which each host is aware of the others), allows us to specify a distributed policy suitable for larger systems,

while keeping a high granularity. Such a policy is distributed at the host level in each *group*, and defines the legal interactions between processes running on different hosts. It states how pieces of authorized information may be accessible by applications and users from any given host of a group.

The model and implementation that we present in this thesis focus on the *confidentiality* and *integrity* aspects of information. Attacks against *availability* are not covered by our approach. We only use off-the-shelf components on commodity hardware, and the only trusted code is our modified operating system kernel.

The reminder of this thesis is organized as follows. The first part introduces the context of this research. It first presents the necessary background in terms of access control, firewalls and information flow control. After this, related work in the field is reviewed and compared to our approach. The second part focuses on intrusion detection in isolated hosts. It presents our model of intrusion detection based on taint marking and its implementation. The last part presents the extensions of our model and implementation to detect intrusion detection in network and distributed environments, as well as our experiments and results.

Part I

Research Context

Chapter 1

Background and Related Work

This Ph.D. project focuses on detecting intrusions at the operating system kernel level, based on an information flow tracking model implemented on top of access control mechanisms (the Linux Security Modules). These three aspects are central to our approach, therefore, this chapter provides an overview of the background literature in these fields. Access control is first introduced, opposing traditional discretionary access control coming as standard with most operating systems, with mandatory access control as implemented in SELinux amongst others. Classic information flow control models are then introduced, followed by modern decentralized approaches as well as related work in terms of information flow tracking and taint marking. Finally, an overview of existing research in the field of intrusion detection is presented.

1.1 Traditional security mechanisms

When it comes to secure information systems, firewalls and access control provide basic security by enforcing OS and network level security properties. These are available in most if not all operating systems. The first part of this chapter is dedicated to these mechanisms, and highlights their shortcomings with respect to the problem we aim to address.

1.1.1 Firewalls

Firewalls are devices or software that filter network traffic at different layers of the ISO network model. They can be set up to restrict access to a personal machine or a company's network from other untrusted networks, thus creating trust boundaries [35]. Individuals can use software firewalls on their personal/portable computers to define and enforce policies concerning both incoming and outgoing network traffic.

Deep Packet Inspection (DPI) firewalls identify anomalous patterns in traffic volumes by inspecting both the headers and content of packets. They provide the capability of identifying anomalous network traffic as well as managing normal traffic. They also form the core of many commercially-available firewalls and intrusion detection systems (IDS). Tamer *et al.* [1] present a survey of the Deep Packet Inspection algorithms, implementation techniques, research challenges and their usage in several existing technologies for intrusion detection systems. Some of the highlighted challenges include the complexity of research algorithms, the ever-increasing number of attack signatures (which negatively impacts on performance) and the increasing prevalence of encrypted data which DPI cannot examine.

Considering the problem we seek to address, that is, detecting intrusions in potentially complex distributed systems, firewalls have several limitations:

- Regular (*i.e.* non DPI) firewalls filter traffic based on reduced sets of properties, extracted from packets headers. This is not suitable when dealing with advanced security policies.

- DPI firewalls can be used to analyse network traffic in a more fine-grained manner, however, since both the packets headers and packets content are analyzed, the overall process implies high performance overhead.

1.1.2 Access control

Access control is the fundamental security mechanism of all operating systems. Though the generic concept of access control exists in many forms, and may be applied to any kind of resource (*e.g.* databases, web content *etc.*) our primary focus in this thesis is operating system security. Amongst the available variants of access control, Discretionary Access Control (DAC), Mandatory Access Control (MAC) and Role Based Access Control (RBAC) are most commonly implemented in commodity operating systems. The following first introduces the notion of access control policy, along with the various mechanisms to represent it, and then presents those three access control variants, as well as implementations of MAC in modern operating systems.

Access control policy

When setting up a system, it is important to clearly understand the security requirements that are involved, and to list them explicitly. This is done by specifying a policy. It is defined at a high level of abstraction, and it represents a concise and formalized set of goals and requirements [2]. In the case of access control, the security policy (access control policy) defines how subjects (*e.g.* users or processes) are allowed to access objects (*e.g.* files), by specifying a set of authorized operations (*e.g.* read, write). Common representations of such policies include Lampson's matrix, access control lists and capabilities.

In 1974, Lampson described an access control matrix [41]. It is a table indexed by subject and object (Lampson uses the term *resources*). The cells of the matrix contain access attributes that specify the kinds of access each subject is allowed to perform on each object. Figure 1.1 shows an example of access control matrix.

	/etc/passwd	/etc/apache2.conf	/var/log/messages
Alice	{read}	{write}	{read}
Bob	{}	{write}	{read}
Carol	{read}	{}	{read}

Figure 1.1: Access control matrix

For each object, the corresponding column lists all the kinds of access any subjects have to that object.

Access Control Lists (ACL) associate each object with an access control list, which is a column in Lampson's matrix. ACLs are the most common way to represent access control authority relationships in modern operating systems. An ACL specifies which subjects are allowed to operate on the object, as well as which operations are permitted. When using ACLs, objects are identified by path names and other *forgeable*[1] references. On UNIX, ACLs contain an *owner*, a *group*, and rights in (R,W,X) standing for *read*, *write*, *execute* respectively. Different rights can be assigned for the owner, the group and the other subjects.

Another way to express an access control policy is to use capabilities. A *Capability* is a communicable and (assumed) unforgeable token of authority. A user or process that possesses a capability will have the right to access certain objects, as described by this capability. Processes can perform some operations on capabilities such as deleting them, passing them to another process or transforming them into less privileged ones. Capabilities are implemented as privileged data structures residing in kernel memory. A capability system associates each subject with a list of capabilities

[1] Such a reference does not give any information about who holds it and which access rights are associated with it.

(also called C-list) which can be represented as a row in Lampson's matrix. However, Miller *et al.* [49] claim that capabilities based models have dynamic aspects that cannot be represented in Lampson's matrix, as it is only a static representation of access rights. Miller *et al.* show that capabilities systems are actually more sophisticated than access rights, and that a direct comparison using Lampson's matrix is not accurate. It should be emphasized that "Portable Operating System Interface for Unix" (POSIX) capabilities are a different kind of capabilities, and are not associated with any object. A process owning a POSIX capability will have some privileges associated with some operations, like listening to ports under 1024 which normally requires root privileges. It is a coarse grained approach aimed to parcel the power of the root user, avoiding the use of *setuid*.

Discretionary access control

Discretionary access control (DAC) is the most commonly used access control mechanism and is the default on UNIX based systems. Access is restricted given the *identity* and the *group* of subjects trying to access objects. It is said to be discretionary because subjects are able to transfer certain permissions to each other at their own discretion. This involves security related issues in systems where end-to-end security policies need to be enforced.

Role Based Access Control

With Role Based Access Control (RBAC), the permissions to perform operations are assigned to specific roles. Permissions are not directly assigned to subjects, but to roles instead. It differs from the ACLs and allows finer grain management of user rights. User rights are managed in a way that has a meaning at the application or OS level, rather than using low level attributes. It has been shown to be a good model for implementing the *separation of duties*.[2] Each subject and object has a set of *security attributes*, and any operation requires to test that it conforms to the policy. It is therefore a particular case of mandatory access control.

Mandatory access control

Mandatory access control (MAC) is based on authorization rules (policy) enforced by the operating system, that are not modifiable by users (it is not discretionary). The Trusted Computer System Evaluation Criteria (TCSEC) [55], also known as Orange Book, defines MAC as "a means of restricting access to objects based on the sensitivity (as represented by a label) of the information contained in the objects and the formal authorization (*i.e.,* clearance) of subjects to access information of such sensitivity". A later publication from the NSA [43] states that this view of MAC is tightly coupled with Multi Level Security (MLS, see Section 1.2.1), and is insufficient to meet the needs of either the US Department of Defense or private industry as it ignores critical properties such as intransitivity and dynamic separation of duty. In response, the NSA proposed a more general notion of mandatory security that was first introduced by the Secure Computing Corporation [15]. A mandatory security policy is considered to be any security policy where the definition of the policy logic and the assignment of security attributes is tightly controlled by a system security policy administrator [43].

Generally speaking, mandatory access control policies are expressed in terms of security labels attached to subjects and objects [62], as is the case with MLS systems. A label on an object is called a *security classification*, and a label on a subject is called a *security clearance*.

With MAC, regular users cannot change the classification of information, and the policy is enforced by the operating system at the kernel level (see the following subsection about MAC frameworks). Some works have been focusing on the verification of the policy consistency against a given set of security goals [12, 38]. By using MAC mechanisms, one can finely control the operations each subject is allowed to perform on the objects of the system. When configured correctly, those mechanisms can significantly improve security by rejecting illegal accesses that would have been allowed otherwise.

[2] Also known as *segregation of duties*, it is a concept of having more than one person required to complete a task.

MAC frameworks

Advances in common operating systems include the improvement of access control mechanisms. While traditional discretionary access control remains widely used, previous research on mandatory access control has led to implementations in common operating systems, such as Linux, FreeBSD, MacOS X and Windows. Examples include SELinux [64], AppArmor [54], Smack [63], Tomoyo [30] for Linux, and TrustedBSD [70] for FreeBSD. When used in so-called *enforcement mode*, they block illegal accesses to objects. When used in *permissive mode*, their behavior is comparable to an intrusion detection system, and alerts are logged when the policy is violated.

The following describes SELinux and AppArmor in further details.

SELinux [64] emerged from research led by the National Security Agency of the USA. It is the first security module available in Linux, and it has been designed to implement a flexible MAC mechanism called *domain and type enforcement* (DTE). Domain and Type Enforcement (DTE) has been presented [DTE95,DTE96] as a model that improves access control. DTE groups processes into *domains* and files into *types*. It restricts access from domains to types as well as from domains to other domains. The kinds of access modes that are considered by SELinux can be any of read, write, execute, create, and directory descend. Domain access refers to the right to send signals as well as to transition to a new domain. At any given time, a process belongs to exactly one domain. Transition to a new domain is done by executing a file which has been defined in the policy as an entry point to the new domain. There are three types of domain transitions: `auto, exec, or none`. For instance, if a domain D_A has auto access to another domain D_B, and a process in D_A executes an entry point for D_B, it will automatically switch to D_B. The `exec` property is similar, except that the process can *choose* whether to switch to a new domain or not, by executing one of its entry points.

AppArmor [54] is a simple MAC implementation available in the Linux kernel as an alternative to SELinux. AppArmor aims to be easier to use and configure than SELinux. It is used by default by Novell in their products and comes with a predefined policy, and a set of generic definitions to ease the difficulty of creating new policies.

A significant amount of work has been done on defining default security policies for SELinux and AppArmor, offering rules for many server applications interacting with insecure data coming from unknown clients through network connections. This makes those tools valuable for system administrators, reducing the work needed to set up complex security policies in real life systems.

Distributed MAC

With the growing number of distributed environments and services across the internet, especially during the last decade, researchers have focussed their interest on the extension of mandatory access control [46, 33, 72] policies to distributed systems so as to control interaction between applications of multiple hosts.

1.1.3 Limitations of access control

Access control, and especially MAC systems are useful to enforce strict policies, dramatically improving the security of operating systems. As compared to traditional discretionary access control, MAC offers tight control over access to objects by subjects or processes, in a centralized fashion. However, access control focuses on the access to resources (*i.e.,* system objects containing information), rather than information, and does not make any distinction between the two. Information flow control and taint marking models allow for more flexibility. The next section further discusses these aspects.

1.2 Information flow control

Contrary to access control policies, which enforce security policies by controlling access to objects containing information (which we call *containers of information*), information flow control focuses on the information itself. Thus, rather than preventing illegal (direct) access to containers, it prevents illegal (direct or indirect) access to information, by specifying a policy regarding information

flows between classes of information. This is a key difference between access control policies and information flow policies. The term *taint marking* is often used to refer to models of information flow, where *taint data* is propagated in *labels*. Information flow tracking models do not enforce a policy, but rather observe information flows and report illegal actions.

	file 1	file 2	file 3
Alice	{read}	{write}	{read}
Bob	{read}	{read}	{}

Figure 1.2: Example: Access control rights

Figures 1.2 and 1.3 illustrate an example of illegal information flow. Figure 1.2 defines the access control rights for Alice and Bob on three files of the system, in the form of an access control list. Figure 1.3 shows how an illegal flow is possible by indirect interaction between Alice and Bob: Bob is able to access information he shouldn't have access to. Bob does not have the right to read file 3, but Alice does. Alice reads file 3 and writes its content to file 2. Bob has read access on file 2. This example highlights the key difference between access control and information flow control: access control does not prevent indirect access to information. Enforcing an information flow policy would have prevented bob to access file 2.

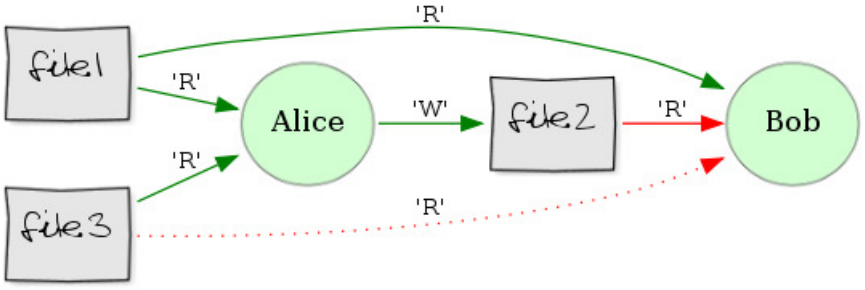

Figure 1.3: Example of illegal indirect flow.

1.2.1 Multi Level Security

The following presents the most common approaches of *multi-level security*. Though such models can be considered as MAC to some extent, these enforce information flow policies. Therefore, we qualify them as information flow control rather than access control.

In 1973, the Bell-LaPadula model was introduced [42], with the primary goal of protecting confidentiality. It is also known as Multilevel Security, and systems that implement it are called Multilevel secure or MLS systems [2]. In this model, subjects and objects are labeled with a security level, which represents their sensitivity or clearance. Any information flow from a high security classification to a lower security classification is illegal [4, 19, 23]. Implementations of MLS try to accurately observe data manipulations in order to prevent illegal information flows. Operating systems with MLS implementation include SELinux, FreeBSD, Solaris and BAE XTS-400.

In 1976, Denning introduced "a lattice model of secure information flow" [17]. She defined it as a mathematical framework suitable for formulating the requirements of secure information flow among security classes. This formal model involves objects, processes and a set of security classes. Objects each belong to a security class, subjects are objects, and processes are the active agents responsible for all information flows. The set of security classes encompasses the concepts of security classifications. Denning also introduces a "flow relation" and the "class combining operator", which together with a set of security classes forms a Lattice.

In 1977, the Biba model [7] was introduced, protecting integrity. It is often viewed as the Bell-LaPadula model upside down [2]. It defines the Biba integrity property as follows: a high integrity

process cannot read lower-integrity data, execute lower-integrity programs or obtain lower-integrity data in any other manner.

In 1987, Clark and Wilson proposed the Clark-Wilson integrity model [14]. As opposed to Biba, it is not a direct derivative of the Bell-LaPadula model, and it does not use label based classification. It is derived from a concept of "double entry bookkeeping" an old practice used in accounting [2]. In this model, low integrity data can flow to high integrity data only if it goes through a *Filter* (an information flow channel). Clark and Wilson also claim that the security needs in the commercial area are as important as those of the Department of Defense.

The models of Bell-LaPadula, Biba and Clark-Wilson can be represented as Lattice models in Denning's framework. Furthermore, combining the Biba and Bell and LaPadula models results in a Lattice, as lattice-based information flow policies that combine several lattices can be cast within a single lattice [62].

The Chinese wall model introduced by Bewer and Nash in 1989 [6] is a hybrid security policy that addresses both confidentiality and integrity. The motivation behind the Chinese wall policy is to group datasets into "conflict of interest classes". In such a model, the subjects can access at most one dataset belonging to the same conflict of interest class. A common example to illustrate this model is the example of consultants dealing with confidential company information for their clients. A consultant should not have access to the information of two concurrent banks, or two concurrent companies of the same sector because it would create a conflict of interest and affect the way the consultant behaves. There is a dynamic aspect with the Chinese wall policy: before a consultant actually accesses confidential information from a specific company, say a bank company, he is allowed to access the information of any bank company. As soon as he has accessed the information from one bank, he cannot access any information from any other bank.

1.2.2 Decentralized models

In 1997, Myers and Liskov proposed a decentralized model for information flow control [50]. This model applies to systems with mutual distrust and decentralized authority. It differs from multi-level security models by allowing users to declassify information in a decentralized way and improves support for fine-grained data sharing. This model allows users to associate confidentiality and integrity labels with data and to restrict information flows based on these labels.

With MAC systems, an administrator sets a system-wide policy. When a server runs multiple third-party applications, it is difficult for a central administrator to understand in detail the security of all the applications. With Decentralized Information Flow Control (DIFC), the policy is partially delegated to the individual applications [40]. Flume, Asbestos and Histar [40, 76, 21] are implementations of decentralized information flow control at the operating system level. Flume [40] has been implemented in Linux and uses the standard operating system abstractions commonly found on UNIX systems (processes, pipes, etc.). In Flume, processes are confined according to a flow control policy. Histar [76] is an operating system aiming to minimize the amount of code that must be trusted. It provides a secure operating system using mostly untrusted user-level libraries (the only fully trusted code being the kernel). It uses Asbestos [21] labels on six OS level object types (threads, address spaces, segments, gates, containers and devices).

1.3 Related work

In the previous sections, we have shown how information flow control addresses the problem of tracking *indirect* information flows within a system. Our work uses such mechanisms so as to track information flows at the operating system level. Recent work have been focussing on information flows control and information flow tracking at different levels for malware analysis, detect privacy violations or to enforce complex security policies. These include VTT *et al.*'s model, Panorama [73], TaintCheck [53], TaintDroid [22], Laminar [25], Pedigree [74], Aeolus [13] and DStar [77]. This section first presents each approach individually and then compares them together.

1.3.1 VTT model

In 2009, Valérie Viet Triem Tong (VTT) *et al.* [66] proposed a model for specifying and enforcing a fine-grained information flow policy. This model relies on tainting techniques in order to provide information flow tracking commodities. Content and containers are distinguished: content are pieces of information while containers are logical storage objects such as files or memory pages. Information flows are observed using tainting techniques. Tainting is performed by propagating tags: containers are each labelled with two tags, an *information tag* describing the current content of the container and a *policy tag* defining the policy regarding the information that can flow towards this container. Content and policy are described in such tags at any stage and for any supervised[3] container in the system. The information flow policy can either be automatically constructed from a DAC policy or configured by an administrator. VTT *et al.*'s model is used as a basis in the work presented in this thesis. We will come back to it later in the next chapter for a comparison with classic multilevel security models as well as decentralized information flow control models. The reminder of this section presents recent information flow models and how these differ from our approach.

1.3.2 Panorama

Panorama [73] is a system-wide information flow tracking model based on dynamic taint analysis, focussed on detection and analysis of malware for Microsoft Windows. It combines taint propagation information at the hardware level with operating system knowledge, so as to generate *taint graphs*. Such graphs represent information flows made by processes on tainted information, and help identify how information is propagated in the system. Using such taint graphs along with a policy allows for automatic detection of malicious code. Panorama provides a fine-grained information flow analysis, involving a small number of *false positives*. It suffers from a significant slowdown of 20 times in average. However, given the purposes of such an analysis, this performance overhead is not considered as a severe limitation. Automatic detection is done in three steps, *test*, *taint* and *analyse*. A test engine first runs series of automated tests. Then, a taint engine monitors how sensitive information is propagated within the system. A malware detection engine along with a set of policies is able to detect malicious code. Finally, a malware analysis engine can be used to examine the taint graphs, and provides detailed analysis information. Panorama was implemented on top of QEMU, for processor emulation, along with a kernel module called *module notifier*, loaded on the guest Microsoft Windows operating system. As compared to our current work (based on VTT *et al.*'s model), Panorama differs in the sense that it provides finer granularity when observing information flows, but it also involves a high performance penalty, and requires hardware emulation, which differs from our objectives, presented in Section 2.3.

1.3.3 Taintcheck

TaintCheck [53] dynamically taints incoming data from untrusted sources (*e.g.* network) and detects when tainted data is used in any way that could be an attack. It uses full system emulation at the instruction level so as to provide a very fine-grained approach. However, as with Panorama, the main limitation of such instruction-level models is a very high penalty in terms of performances; a slowdown of 1.5 to 40 times is to be expected, according to its authors. For the same reasons, this approach is not in accordance with our objectives.

1.3.4 Argos

Argos [58] is an emulator, based on Qemu, for generating signatures of attacks automatically. It observes information flows in the guest (emulated) system so as to track illegal use of unsafe information, such as information from the network. Information from unsafe sources is tainted with *tags*. Such *tags* are attached to the memory at the byte granularity, and to CPU registers using a single tag per register. Argos traces access to physical memory addresses, and generates logs when

[3] Supervised containers have a policy tag, non supervised containers eventually obtain an information tag as these get tainted.

a violation is detected. Such logs contain registers and memory information (memory dumps), and are used for automatic generation of signatures (in Snort rules format) as well as manual analysis. Argos is able to detect attacks in userspace as well as in kernelspace. When an attack occurs, Argos injects its own shellcode, using the address space of the attacked process, so as to gather additional information from this process. Such information may for instance be transmitted to the host (running the emulator) for forensics analysis. In order to generate signatures, Argos looks for patterns by comparing the memory dumps and the traffic generated by the attack (after filtering out useless information, such as traffic on untargeted ports). As for Taintcheck and Panorama, the objectives of our work, presented in Section 2.3, differ from these of Argos.

1.3.5 Taintdroid

TaintDroid [22] is an information flow tracking system for realtime privacy monitoring on smartphones. It is based on taint marking at four different levels of granularity, respectively at the variable, message, method and file levels. TaintDroid has a performance overhead of 14% on the CPU. This approach is similar to the approach we have taken in this current work. However, TaintDroid is focussed on the Android platform using the Dalvick interpreter and therefore it does not apply to native applications, which represent most of the software present on standard desktop and server operating systems. Furthermore, it does not propose a fine-grained information flow policy, but rather focusses on some specific data with respect to privacy issues.

1.3.6 Laminar

Laminar [61] is a hybrid solution combining language level and operating system level Decentralized Information Flow Control (DIFC). It requires light modifications (less than 10%) in the code of the programs, where programmers can use *secrecy* and *integrity* labels so as to express security policies. It uses the same abstractions for OS-level resources, and heap allocated objects. It implemented as a modified Java virtual machine along with a Linux security module. The performance overhead of this approach varies from 1% to 56%. While Laminar offers interesting results by combining several approaches, it requires modifications in the code of applications, where our approach focusses on the use of unmodified applications on commodity hardware.

1.3.7 Pedigree

Pedigree [74] enforces information flow control across a network for legacy applications and operating systems. It implements two functions: a trusted *labeller* and a central *controller*. The trusted *labeler* propagates labels on each host, it runs as a trusted module at the operating system level, and tracks information flows at the level of files and processes. The central controller enforces the policy. Therefore, the so-called *data plane* (forwarding of labels) is separated from the *control plane* (enforcement of the policy). The security model of Pedigree is based on a lattice, and the policy is centralized. 64-bit labels are attached to each resource (*i.e.*, files or processes) and contain *taint*. On each host, a *label store*, implemented as an in-memory structure, attaches labels to resources. A global *label store* is also maintained, and used by a *network enforcer* to enforce information flows between different hosts. *Taint* may be of two kinds: secrecy or integrity. Users are allowed to create new taint, modify a taint that they own, and modify labels on a resource that they own, based on their *capabilities*. A *capability database* manages the capabilities, and users can have the capability to set or unset the secrecy bit of a taint (s^+ and s^-), to set or unset the integrity bit of a taint (i^+ or i^-) and to add or remove users who may manage the capabilities of a taint (o^+ or o^-). The main difference of our approach as compared to Pedigree is the information flow policy itself. We compare Pedigree with our approach in Section 6.6.

1.3.8 Aeolus

Aeolus [13] is a platform for building secure distributed applications. It performs decentralized information flow tracking at the *thread* level. Similarly to other models of DIFC, it allows users to define restrictions on the use of their own information. It is based on simple rules involving

principals and *tags*, where *tags* are used to categorize information, and *principals* are the entities interested in such information. It provides fine-grained delegation of authority, and supports *revocation*. It makes use of a memory-safe language to isolate threads from each others. Support for distributed programs involve a RPC mechanism, and provides the concept of *boxes*, allowing confidential information to be communicated between two ends without tainting intermediates which do not observe the information flow. Aeolus is OS-independent, and it is implemented as a set of runtime libraries. Its main implementation supports Java, but it has also been ported to C# and PHP. Contrary to language-based information flow tracking systems, Aeolus does not observe individual variables. It remains more fine-grained than OS approaches, as it observes individual *threads*. A comparison of our work with Aeolus is presented in Section 6.6.

1.3.9 DStar

In the field of decentralized information flow control, Zeldovich *et al* extended their previous work [40, 21, 76] with DStar [77], so as to control information flows in distributed systems. Dstar controls how information flows between processes on different machines. It provides DIFC mechanisms for use by applications, in order to define legal interactions between mutually distrustful components. By opposition with MAC, where a central administrator controls the system, DIFC gives control to application programmers, leading to a finer granularity. In DStar, labels are attached to processes, and define the legal behavior of processes. By using such labels, Dstar ensures that only processes allowed to communicate may do so. Each label contains a set of two categories: *secrecy* and *integrity*. Secrecy categories in a message determine who is allowed to receive it, and integrity constrains who may have sent it. It follows a "no read-up, no write-down" logic, with respect to a partial order between labels, defined by the "can flow to" ($\sqsubseteq$) function. It ensures that untrusted code does not access inappropriate data. In DStar, each process also has a set of privileges, which allow it to bypass some permissions that are normally forbidden by the $\sqsubseteq$ relation between labels. Processes may also raise their own label given their *clearance*. When processes own a category, these can raise the labels of other processes in that category. In order to carry labels over the network, DStar uses so-called *exporter daemons* on each host, which are the only processes sending or receiving DStar messages over the network. Trust is decentralized between categories owners of each host, through local exporters. Trust between exporters relies on cryptographic certificates, and exporters may delegate trust in a category to other exporters. As for Pedigree and Aeolus, we propose a comparison of our work with DStar in Section 6.6.

1.3.10 Comparison of related work

Current information flow control and information flow tracking models can be categorized into three types: language level, operating system level and architecture level [61]. Language level solutions [50, 51] rely extensively on type system changes and modify the program structures. Such solutions are not able to track security violations at the level of system objects (such as files and sockets). Operating system level solutions [40, 76, 21] rely on page mappings and OS-level abstractions, and cannot accurately monitor information flows into applications, as those do not have access to inner data structures [61]. However, these are able to observe information flows over all the system. Our work follows this approach, as presented in more details in Section 2.3. Architecture level solutions [67, 78] are able to track data labels within applications but require trusted software to manage the labels and involve high performance penalties in the case of full system emulation.

In terms of performance overhead, Taintcheck and Laminar have high performance penalty due to their low-level approach (full system emulation). Though this provides a fine-grained approach while observing information flows (which provides interesting results for malware analysis) this approach is not practical for runtime monitoring of a full operating system, as required by our intrusion detection approach (our requirements are explained in more details in Section 2.3).

Laminar, Pedigree, DStar, Aeolus and DIFC models *enforce* a security policy (*i.e.,* these block illegal information flows) while Panorama, Taintcheck, VTT *et al.* and Taintdroid *taint* information and let illegal information flows occur. For a comparison of our approach with these related work, see Sections 3.9 and 6.6.

	Implementation	Performance overhead	Distributed	History
Panorama	OS and architecture	high	no	no
Taintcheck	architecture	high	no	no
Taintdroid	OS/language	low	no	no
Laminar	OS and architecture	high	no	no
Pedigree	OS	low	no	no
Aeolus	Language	low	yes	no
DStar	OS/language	low	yes	no
VTT *et al.*	OS/language	low	no	yes

Figure 1.4: Comparison of related work

Figure 1.4 compares recent approaches of information flow control and tracking, with respect to the following criteria:

- *Implementation* refers to the level (*i.e.,* layer) of deployment of the approach. *OS* refers to operating system level approaches (userspace libraries wrappers or kernel), *architecture* refers to full system emulation, and *language* refers to the modification of virtual machines or interpreters, or instrumentation of the code of applications.
- *Performance overhead* is a rough estimation of the performance of each approach.
- *History* refers to the fact of keeping tracks of individual pieces of information throughout the system.
- *Distributed* refers to mechanisms providing information flow control or tracking across multiple hosts over a network.

We believe that strict policies are not practical in all situations, as these can potentially break functionalities by blocking legitimate information flows when the security policy is too strict, or, on the contrary, allow illegitimate access when the security policy is too permissive. This becomes particularly problematic when applying such mechanisms to complex distributed systems made of heterogeneous hosts, using multiple applications with various security requirements altogether. Furthermore, most systems use *off-the-shelf* components and applications, and these do not come with predefined policies designed by the developers. Instead, the security requirements are specified *a-posteriori*, which requires a lot of effort and leads to complex security policies. On the contrary, tainting information without blocking allows for information flow tracking. To this extent, VTT *et al.*'s model differs from existing tracking models as it provides fine-grained information flow tracking and keeps information flow history. Its policy differs from traditional MLS and from DIFC (this aspect is further developed in Chapter 2) as it allows to specify rules for individual pieces of information within a system. It also differs from recent approaches, as these are either based on *non-interference* between security levels (*e.g.,* low/high) or lattice-based (usually representing hierarchies of security classes such as *secret, top-secret etc.*). We used the model introduced by VTT *et al.* as the basis of our work. It is presented in more details in Chapter 2.

1.4 Intrusion detection

Intrusion detection is the process of monitoring and analysing system and network events, looking for signs of intrusion. Intrusion detection systems (IDSes) are software layers which automate these monitoring and analysis processes [16]. IDSes are used to detect attacks such as viruses and malicious users or to monitor the security of a system to help in diagnosis and correction of flaws [24].

1.4.1 Host-based and network-based IDS

Host-based Intrusion Detection Systems (HIDS) have access to the operating system information [16, 3]. Such IDSes are able to detect the presence of malware and targeted attacks by analysing low level system objects and information. Furthermore, encrypted network attacks can also be detected by analysing low level network traffic once it has been decrypted. HIDSes can generate alerts corresponding to each malicious system event.

Network Intrusion Detection Systems (NIDS) can monitor segments or sections of networks, depending on their placement [16]. Those typically work in so-called "promiscuous mode" (only capturing traffic) and have very little impact on the network. Such IDSes can consume system resources when dealing with large or busy networks [38].

1.4.2 Anomaly detection and misuse detection

Among existing intrusion detection systems, two major approaches are used in order to differentiate normal behavior and misuse. [82]. Anomaly detection defines a legal behavior that is known to be safe. Any unknown action is considered as illegal. Statistical models are often used in this case. Misuse detection, also called knowledge based, defines what is illegal, based on signatures of misuse actions.

Misuse detection, is the most popular approach amongst commercial IDSes [80]. Misuse IDSes make use of knowledge about known attacks, exploits and vulnerabilities and analyse system events and network traffic looking for matching patterns. Such knowledge is often called signatures. One of the drawbacks of this approach is that the signatures database has to be maintained up to date in order to be effective [16] . Another drawback is that it is possible to forge fake matching patterns in network traffic and/or system events, leading to false positives and overloading of the IDS. Also, such IDSes can only detect known attacks that are already present in the signature database.

Debar *et al.*[16] have shown that misuse detection can be achieved using different methodologies. These include expert systems, signature analysis, petri nets and state-transition analysis. A common method amongst commercial IDSes is the use of signature analysis along with patterns of attacks reduced to a low level of semantics. Well-known misuse detection IDSes include Snort [59] and Bro [57]. These are both open-source.

Anomaly detection IDSes aim to identify abnormal/unusual behaviour (anomalies) by comparing current behaviour to a known normal state. It was first introduced by Denning in 1976 [18]. Denning was assuming that traffic generated by attackers is clearly different from normal traffic, which is recorded into profiles. One advantage of anomaly detection systems is their ability to detect previously unknown attacks (zero day) [56], which attackers may seek to exploit before patches are released to fix the targeted vulnerabilities. Another advantage is the ability to detect different forms of the same attack, where signature-based IDS do not always have all the possible matching signatures [56, 68].

Anomaly detection IDSes rely on several methodologies. Self learning systems (time series based, such as artificial neural networks (ANN), or non time-series based such as descriptive statistics and rule modelling) learn by example what constitutes the normal behavior of a system [3]. Programmed systems are taught by an administrator to detect abnormal behavior. Those can be based on descriptive statistics algorithms, or on a default deny approach, stating only what is legal.

Statistical based anomaly detection models use statistics from different parameters [24]. As stated by Gates and Taylor [25], most modern anomaly detection systems are based on Denning's assumptions [18]. Those assumptions consider that attacks are rare (as compared to normal events) and differ from the normal behavior of the system.

Hybrid systems [56] are combining both misuse detection and anomaly detection approaches.

1.4.3 Policy-based IDSes

Policy-based IDSes are anomaly detection IDSes following a "default-deny" approach. A number of previous works exist in this domain, using sandboxing mechanisms at the language level [36] or via Kernel based reference monitors such as BlueBox, REMUS, LIDS and Ko *et al.* system wrappers [11, 28, 5, 39]. Similar sandboxing mechanisms also exist in user space, namely system

introspection [69, 37]. Blare [81, 82, 66, 26] is an IDS deployed at the host level, and at the Java Virtual Machine level. It relies on information flow models developped in the ISSN[4] team at Supélec. Its first model is host-based and was developed by Jacob Zimmermann [81, 82]. It relies on the principle of *non-interference*. This principle was introduced in 1982 by Goguen and Meseguer, and extended in 1984 by the same authors [27]. It is a strict multilevel security policy model, where information is gathered in isolated security classes. Information cannot flow from one security class to another. Hiet, Viet Triem Tong, Morin and Mé have used the first version of the Blare model along with JBlare[5] to control the legality of information flows in Java programs using a non interference policy. This hybrid intrusion detection (OS/Language levels) allows to refine information flow tracking, thus reducing the number of false positives [34].

1.4.4 Distributed IDSes

Even though distributed systems have become very popular, particularly since the explosion of cloud infrastructures, little research focussed on new models of intrusion detection suitable for such environments. Existing approaches are based on aggregation or centralization of events reported by individual *misuse* IDSes, such as Snort [59] or Bro [57]. Examples of this are the following approaches. In [60], Roschke, Cheng and Meinel proposed and implemented an extensible IDS management architecture, providing central management of several sensors. It integrates several sensors through an *event gatherer*, with support for several implementations of well known IDSes. In [45], Mazzariello, Bifulco and Canonico proposed an approach of *misuse* detection for an opensource cloud computing environment. It targets denial of service attacks, and it is based on Snort [59] for analyzing network traffic.

To our knowledge, the approach that is the most closely related to our current work is an approach of *anomaly detection* introduced by Zimmermann and Mohay in [83]. It focuses on detecting intrusions in clusters based on the principle of *non-interference*. Objects of the operating system are supervised by monitoring the invocation of their *methods* (*i.e.* actions such as `read` or `write`) and producing a *trace*. The policy specifies the legal behavior of the system, by associating *domains* to object *methods*. Violations of the policy are detected by evaluating a trace of the system using an *unwinding* theorem. Such a theorem makes it possible to identify the set of existing traces matching the desired non-interference properties. Reference monitors are deployed on each node of the system, and messages between nodes are instrumented.

[4] Now CIDre.

[5] JBlare is an implementation of Blare in the Java Virtual Machine (JVM) able to monitor information flows withing Java programs.

Chapter 2

Information Flow Models

Our research is the continuation of previous work in the ISSN (Information Systems Security and Networks) team at Supélec (now CIDre). Models for dynamic information flow tracking have been previously proposed, and have shown to be successful in detecting intrusions [66, 82]. Our model is an extension the VTT model, and we target intrusion detection in both isolated and distributed environments. In this chapter, we first present the VTT model (introduced in Section 1.4.3). Then we compare it with existing models of information flow control and present a summary of the properties offered by each model. We finally present our requirements for intrusion detection.

2.1 VTT model

The following is a description of the model introduced by Valerie Viet Triem Tong *et al.* [66] in 2009. This model is an information flow model based on taint marking techniques along with an information flow policy, it protects both integrity and confidentiality. Objects of the operating system potentially containing information, such as files, are called *containers of information*.

Definition 1. Labels called *tags* are attached to each *container of information*. Tags contain meta-information, that are used to describe real content. These tags include a *policy tag*, and an *information tag*:

- The *information tag* represents an over estimate of the information that the container may contain.
- The *policy tag* represents the information flow policy for the container (*i.e.* which information it can legally contain).

Any information flow towards a container, making changes to its content, requires an update of its information tag so as to match the new content. After this, a legality check is performed in order to ensure that its policy (as defined in its *policy tag*) has not been violated. If a violation of the policy occurs, an alarm is raised.

2.1.1 Policy

The policy in the VTT model differs from other information flow models. It is decentralized at the container level in the *policy tags* of each specific container, and states *"which information is allowed to be contained in each container"* or in other words *"what can flow towards each container"*. The policy for a container is expressed as a set of sets. Any set (or any subset of it) of the *policy tag*, represents a legal combination of information for a given container, (*i.e.* a legal *information tag*). Therefore, an information flow towards a container is legal if and only if the updated *information tag* of the container after the information flow occurred is included in one of the sets of the *policy tag*.

- The integrity of containers is protected by controlling which subsets of information are allowed to mix together inside the containers (*i.e.* how information may be altered).
- The confidentiality of information is controlled by determining which pieces of information containers may contain (*i.e.* where information may flow).

Definition 2. Let $\mathcal{C}$ be the set of all containers. For any container $c \in \mathcal{C}$,
$itag(c)$ lists the origin of content residing in the container, *i.e.* its *information tag*,
$ptag(c)$ lists the policy attached to the container, *i.e.* its *policy tag*.

Updates of the information tag

When an information flow occurs from a container C_1 to a container C_2, the information tag of C_2 is updated with the information tag of C_1. Its new information tag (*after the flow occurred*) is the union of its old information tag with the old information tag of C_1 (*before the flow occurred*).

$$itag(C_2)_{new} = itag(C_1)_{old} \cup itag(C_2)_{old}$$

Legality of an information flow

Definition 3. An information flow towards a container is *legal* if and only if its *information tag* is included in one of the sets of its *policy tag*:

$$Legal(itag(C), ptag(C)) \Leftrightarrow \exists p \in ptag(C) | itag(C) \subseteq p$$

Example 1. Consider an information flow from C_1 to C_2 with the following tags:
$itag(C_1) = \{1, 2\}$
$itag(C_2) = \{2, 3\}$
$ptag(C_2) = \{\{1, 2, 3, 4\}, \{5, 6\}\}$
The following update on the information tag of C_2 would occur :
$itag(C_2)_{new} = itag(C_2)_{old} \cup itag(C_1)_{old} = \{1, 2, 3\}$
This information flow is legal because $itag(C_2)$ is a subset of one of the sets of $ptag(C_2)$: $\{1, 2, 3\} \subseteq \{1, 2, 3, 4\}$.

With such a policy, the confidentiality and the integrity properties are independent. For instance, the policy attached to a process might have a low level of confidentiality (*i.e.* it would only have access to a small subset of the information on the system), and a high level of integrity (*i.e.* the data cannot mix with other data) at the same time. Empirically, confidentiality and integrity can be expressed as follows, and are compared to the same notions of the Bell-LaPadula and Biba models later in Section 2.2.

- The confidentiality level of a policy tag is determined by the set of different atomic information it allows in a container, regardless of how it allows them to mix together. The more different pieces of information are legal in the container, the higher the level of confidentiality raises for this container. The less information is legal in the container, the lower the level of confidentiality. For instance, a process with a high level of confidentiality may have authorized access to a lot of different pieces of information, and thus have a policy tag allowing it to contain a high number of different pieces of information. The confidentiality level of a container c can be measured by:
$$\left| \bigcup_{p \in ptag(c)} \right|$$
- The integrity level of a policy tag is determined by the combinations of information it allows in a container. The more the information is allowed to mix with other information, the lower the integrity. The less it is allowed to mix, the higher the integrity. For instance, a process with a high level of integrity may not mix its content with low integrity information and would thus have a policy tag forbidding it. The integrity level of a container c can be measured by:
$$\frac{|ptag(c)|}{|\bigcup_{p \in ptag(c)}|}$$

2.1.2 Dynamic aspect

In the VTT model, the set of authorized operations that processes can perform over objects (containers of information) is not constant. It may dynamically change over time: whether a process can access an object depends on the information that it previously accessed. For instance, a process might have the permission to write to a given container until it reads some data that is invalid in this container, either for integrity reasons (*e.g.,* the new data does not have a sufficient integrity level), or for confidentiality reasons, (*e.g.,* the new data cannot be mixed with less confidential information). This notion of dynamic changes in the authorized behavior of processes could be qualified as a dynamic *clearance*[1].

This dynamic aspect can be summarized as follows:

- A policy is expressed on what containers are allowed to contain.
- The content of containers keeps changing (after each information flow).
- The clearance of a process is dynamic in time.

2.1.3 Lattice

The VTT model can be formally represented in Denning's framework, "Lattice model of secure information flow" [17], and under certain assumptions, its components form a bounded lattice. While the demonstration of this aspect is not covered here, we demonstrate this lattice property later in Section 3.7 for our extension of the VTT model.

2.2 Comparison with lattice based models

The following is a comparison of the VTT model with the most common implementations of multilevel security (MLS) systems and policies. We use the terms *security class* to refer to the policy of one or more containers (see Section 3.7).

2.2.1 Chinese walls

The Chinese walls model is centered on the concept of separation of "conflict of interest classes" (see Section 1.2.1). Such a dynamic property can be defined in a VTT policy. Recall the previous example from Chapter 1 with a consultant working for a bank company. A Chinese wall policy could be defined such that once the consultant had access to information from any bank, his or her access to the information from any other bank would be denied.

In the following, we call I_{Bank_k} the class of all the information related to the bank $Bank_k$. Therefore, in a context where N concurrent banks exist, if the consultant has accessed the information from $Bank_1$ (I_{Bank_1}), his or her access to $I_{Bank_k|2\leqslant k\leqslant N}$ is illegal.

Such a policy can be defined in the VTT model by attaching a user policy to the consultant where each subset of the policy concerns the information from one specific bank. The following policy is an example of this:

$$\mathbb{P}_{U_{Consultant}} = \{\{I_{Bank_k}\}_{1\leqslant k\leqslant N}\}$$

The multiple subsets of this policy have a meaning of exclusion: the legal information for user $U_{consultant}$ is defined by at most one of the composing sets of the policy at one time. It can be seen as an *exclusive or* relation between the composing sets, allowing only one set at a time.

2.2.2 Bell-LaPadula

The Bell-LaPadula model labels data with levels of *classification*. It can be summarized as follows:

- The *simple security property* also known as "no read up" states that no processes can read data up from a higher level of classification.

[1] The notion of clearance here is the same as in the Bell-Lapadula model, defining a level of authorization for a subject over an object. See Section 1.2.1.

- The **-property* also known as "no write down" states that no processes may write data down to a lower level of classification.

While this model protects confidentiality, it does not protect integrity. In the VTT model, a process may not read information that is illegal with respect to its policy tag (*i.e.,* not contained in one of the sets of its policy tag). This means that this information is contained in a higher or incomparable security class in the policy's lattice. This is comparable to "no read up" in Bell-LaPadula.

Example 2. A process with policy {{1,2,3},{4,5,6}} may not read a file containing {1,2,3,4}. It may not read a file containing {5,6,7} either. Both are forbidden with respect to the policy. However, {5,6,7} would be allowed if the policy was {{1,2,3},{4,5,6,7}}, which is considered as a higher security class than {{1,2,3},{4,5,6}}in the policy's lattice (presented in Section 3.7).

In the VTT model, a process may not write information to a file if such information is not legal with respect to the file's policy tag (*i.e.,* not contained in one of the sets of its policy tag). The meaning of this in terms of security class is that the involved information is contained in a higher or incomparable security class in the policy's lattice. This notion is similar to "no write down" in Bell-LaPadula.

Example 3. A process may not write information {1,2,3,4} in a file with policy {{1,2,3},{4,5,6}}. It may not write information {5,6,7} either. Both are forbidden with respect to the policy. However, {1,2,3,4} would be legal with the policy {{1,2,3,4,5,6}}, which would be a higher security class in the policy's lattice.

There are however two major differences between the VTT model and Bell-LaPadula. With VTT, information flows are illegal between different security classes with incomparable levels of security. Also, the VTT policy makes it possible to define which information is allowed in which containers, and is thus attached to containers themselves, it does not express any direct classification of the information.

2.2.3 Biba

The differences between VTT and Biba are similar to those with Bell-LaPadula. Similarly to confidentiality, data with the same level of integrity are isolated as those are considered as being different security classes.

Integrity with VTT is protected on a "by container" basis, and given two pieces of information i_1 and i_2, some containers may be allowed to mix them together ($\{\{i_1, i_2\}\}$) while some other containers may not ($\{\{i_1\}, \{i_2\}\}$). The integrity which is protected is the integrity of the container, not the integrity of the information itself.

2.2.4 Clark-Wilson

The Clark-Wilson model protects integrity. As opposed to Biba, it is not based on Bell-LaPadula, and it does not make use of label-based classification. In this model, low-integrity data can flow towards high integrity if it goes through a *filter* (declassification). This model is not based on a lattice. It is not directly comparable to the VTT model in terms of policy.

2.2.5 DTE

DTE stands for "Domain and Type Enforcement" and SELinux is based on it. (See Section 1.1.2). With DTE, a *domain* attribute is attached to subjects, and a *type* attribute is attached to objects. Restrictions apply from *domain* to *type*, and also from *domain* to *domain*.

In the VTT model, the tags attached to processes and containers can be compared to domains and types in DTE. Information flows between two containers are legal if their policy tags allow it. The information tags state which information the containers contains, and the relation between policy tags and information tags can be seen as domain to type or domain to domain in DTE. In the VTT model, this relation is bilateral:

- The relation between the information tag of a process and the policy tag of a file defines if the process is allowed to access this file in `write` mode.
- The relation between the information tag of a file and the policy tag of a process defines if the process is allowed to access this file in `read` mode or in `exec` mode.

However, the changes of domains in DTE have no equivalent in the VTT model. In DTE, executing a binary program may cause a domain switch for the running process, and the new domain can either extend or restrict the rights of the process. In the VTT model, any information flow between a subject and an object may change the information tag of either the subject or the object, thus restricting the policy in one direction: from the subject to the object if the information tag of the subject has been modified, or the other way in the other case.

2.2.6 Myers and Liskov

As in the VTT model, the Myers and Liskov decentralised information flow control model (DIFC) is related to mandatory access control in the sense that the security policies are mandatory, and not enforced at the discretion of application writers [40]. Where the M&L model allows decentralization of the policy with the applications being allowed to declassify information that they own, VTT policy specification is centralized (though future works are planned to provide declassification in the model). Both M&L and VTT are based on a lattice and protect both integrity and confidentiality of data.

2.2.7 Summary of the comparison

The VTT model can be seen as a combination of Biba and Bell-LaPadula as it addresses both confidentiality and integrity aspects at once. It has however a dynamic aspect in common with the Chinese walls. Furthermore, it allows data isolation when security classes of the same level are not directly comparable. This later aspect is comparable with models based on *Multiple Independent Levels of Security* (MILS).

	VTT	**B&LP**	**Biba**	**CW**	**C&W**	**M&L**	**DTE**
Confidentiality	yes	yes	no	yes	no	yes	yes
Integrity	yes	no	yes	yes	yes	yes	yes
Dynamic	yes	no	no	yes	no	yes	yes
Decentralized	no	no	no	no	no	yes	no
Declassification	no	no	no	no	yes	yes	no[1]
Distributed	no	no	no	no	no	yes	no
Content based	yes	no	no	no	no	yes	no
Flow history	yes	no	no	no	no	no	no

Figure 2.1: Comparison of information flow models.

[1]there is no declassification mechanisms in DTE. However, domain transitions may provide comparable properties in some situations. extent.

Figure 2.1 is a comparison of information flow models: B&LP stands for Bell-LaPadula, CW stands for Chinese walls, C&W stands for Clark-Wilson, M&L stands for Myers-Liskov. The declassification aspect of the VTT model is a work in progress in the CIDre team. The term *decentralized* refers to the way the policy is defined. If it is centrally defined by one single authority as it is most often the case, then it is characterized as centralized. *Distributed* refers to the network distributed systems such as web services with multiple hosts. *Content based* refers to the distinction

between containers of information and content. *Flow history* refers to the ability to describe the origins of all the content that is residing in a container.

2.3 Objectives and requirements for intrusion detection

In this Ph.D., we aim to dynamically detect intrusions in isolated hosts as well as in distributed systems composed of multiple hosts. Our objectives are the following:

- Detecting violations of *integrity* and *confidentiality* (which we consider as intrusions).
- Detecting *successful* attacks targeting all kinds of components (applications, OS-level services *etc.*).
- The ability to use *off-the-shelf* components: unmodified applications running on commodity hardware.

Our approach of intrusion detection follows the *anomaly* detection paradigm: we observe illegal information flows within the operating system, with respect to a security policy.

There exists a number of information flow control models in the literature. Some of these models can be used in *permissive mode*, where the security policy is not enforced, but alerts are raised instead. Such behavior allows the information flows to actually happen and modify the state of the system. This is a first requirement for our approach of intrusion detection (we do not aim to prevent intrusions). Another requirement is the ability to track the origin of information residing within any of the objects of the operating system. Where most models of information flow control would let information spread once configured in *permissive mode*, they would not taint information : no tracking of the propagation of information within the operating system would be possible.

As shown in Figure 2.1, the VTT model fits both of these requirements :

- It is a permissive model: it does not enforce the policy, and it does not forbid information flows. Flows happen and modify the state of the system.
- The information flow history is kept, and allows to track the origin of information residing in any container of the system. This aspect relies on so-called *taint marking* techniques. It will be further described later in Chapter 3.

For these reasons, our approach of intrusion detection is based on VTT's model. The contributions of our work are presented in the next parts of this thesis. Our first contribution is an extension of VTT's model and its implementation in the Linux kernel. This is presented in Part II. Our second contribution is the extension of this first work to fit distributed systems, and it is presented in Part III.

Part II

Intrusion Detection at the Host Kernel Level

Chapter 3

Extended Model

As presented in Chapter 2, a number of information flow models exist. These may be applicable to intrusion detection when used in a *permissive* mode, where the policy is not enforced and information flows actually occur even when these are illegal. Our choice of not enforcing the policy is motivated by the fact that we are interested in intrusion detection rather intrusion prevention. However, future research in this field may also focus on the *enforcement* mode of information flow models. As shown in Figure 2.1 in Chapter 2, the VTT model offers properties that best fit our requirements. Therefore, we use it as a basis in our intrusion detection approach. We have however identified some evasion issues when using the model as-is for designing a host kernel level monitor. Although the VTT model offers the properties that are needed for our approach of intrusion detection, it lacks consideration of some aspects of the operating system that are necessary for realistic intrusion detection. This chapter first highlights the evasion issues we found, and then presents our extended model and how it allows to detect intrusions in isolated machines (distributed aspects are covered in the third part of this thesis).

3.1 A model based on VTT

The VTT model provides fine-grained information flow tracking between containers of information. When applied to an operating system of the UNIX family, it allows to track information between objects of the operating system such as files, sockets and the like, and users. This notion of user differs from the traditional UNIX notion: users in VTT are considered as containers. Recall from Chapter 2 that for any container c, $itag(c)$ lists the origin of content residing in the container, which we call its *information tag*. In VTT, this applies to users as well, as information tags are attached to their representing containers. For instance, users A and B would be represented as containers u_A and u_B, with $itag(u_A) = i_A$ and $itag(u_B) = i_B$. If we now consider a container c with the following policy: $ptag(c) = \{i_A, i_B\}$, stating that c may only contain i_A or i_B, or both at the same time (that is, any subset of $\{i_A, i_B\}$), then only users A and B are allowed to write in c (no matter if one of them already wrote information to this container before the other).

3.1.1 Evading VTT

When applied to a real operating system, this model can be evaded through code execution, as it does not confine executable code. Furthermore, the previous notion of users is only theoretical: no process confinement mechanisms are defined in the model. Whereas information from an exclusive list of users is allowed in each container, the reverse is undefined (*i.e.* how information is allowed to flow towards a user). As an illustration of the shortcomings of this model with respect to code execution, consider the following example:

Example 4. : A malicious user exploits a flaw in a service running on a web server, and injects arbitrary code into the process running this service. The injected code is then interpreted and it

writes a malicious script into a new file, before executing it as the current user.

In such a scenario, the VTT model would forbid the process to write into any file c (container) for which access is not allowed to the user u_{web} running the web server (*i.e.* the policy attached to c does not allow u_{web}). On the contrary, writing to any file allowing u_{web} in its policy is authorized, and the same goes with the creation of new files (as in the previous example). In such a situation, there is no way to detect the intrusion: this is one potential scenario of evasion.

3.1.2 Proposed extension

We have presented and published the following extension of the VTT model at ICC 2011 [65]. This new model improves the following aspects :

1. The execution of code and programs is supervised, based on the distinction between active code, that is executed by processes, and passive stored information.

2. Containers of information are considered separately, depending on whether these are stored in memory or on-disk. The former are called *volatile* containers, and the latter are called *persistent* containers. We also make the distinction between (passive) objects, storing information, and (active) subjects (*i.e.* processes running code on behalf of users).

3. The information flow policy can be expressed separately for users, executable code (which we also call programs[1]) and containers.

4. The information flow policy can be derived from a mandatory access control policy. We have formally defined a method for deriving an AppArmor[2] policy into an information flow policy that is applicable to intrusion detection. It remains possible to derive an information flow policy from a discretionary access control policy, as it has been done in previous work with the VTT model.

We further detail these aspects in the reminder of this chapter.

3.2 Data and code distinction

Recall from Definition 1 in Chapter 2 that *tags* contain *meta-information* describing actual information (or data[3]) of the system. In this new model, meta-information is represented by two sets $\mathcal{I}$ and $\mathcal{X}$ as follows:

- $\mathcal{I}$ is the set of all meta-information describing *passive data* (*i.e.,* stored in a file). Note that *executable* code (*e.g., shared libraries, binary programs, executable scripts*) is equally represented in $\mathcal{I}$ as long as it is not *executed*, *i.e.* as long as it is not running as the code of a process. Thus, stored data representing code is represented in $\mathcal{I}$.

- $\mathcal{X}$ is a set describing active code being executed (*i.e.*, being run as code in processes). Each element of $\mathcal{X}$ is an image of one *passive* information element of $\mathcal{I}$, through the *Run* relation defined below.

This distinction of $\mathcal{I}$ and $\mathcal{X}$ was inspired by Denning's assumption: "Processes are the active agents responsible for all information flow" [17].

Definition 4. The execution of code is characterized by the following relation:

$$Run : \mathcal{I} \rightarrow \mathcal{X}$$

[1] In this thesis, we equally refer to *executable code* or *program* to refer to any given combination of executable information, potentially being executed by one or several processes.

[2] As introduced in Chapter 1, AppArmor is a Linux security module developed by Novel.

[3] We both refer to *information* and *data* interchangeably.

$\mathcal{X}$ is a bijection of $\mathcal{I}$ through the relation *Run*. Each program is described by one or more elements of $\mathcal{I}$ when stored on disk, and by their image through *Run* when running as the code of a process. We do not have any a-priori knowledge concerning the *executable* aspect of information. Therefore, each passive information of $\mathcal{I}$ has an image in $\mathcal{X}$, that is used upon eventual execution.

Definition 5. A program (or application) is defined as a set of executable information in $\wp(\mathcal{I})$[4]. We define the set of all programs as Π:

$$\forall \pi \in \Pi, \pi \in \wp(\mathcal{I})^5$$

Usually, we would label each supervised program with a unique meta-information of $\mathcal{I}$, however, in some cases, programs may be composed of multiple elements combined together, *e.g.* a C program linked with shared libraries as in Example 5, or a virtual machine or interpreter loading a script file, as it is the case with most dynamic languages such as Ruby, Python, PHP and many others. In such cases, the final program is the set of all of its composing elements, and it is tainted with multiple meta-information of $\mathcal{I}$. This aspect allows us to define the legal interactions amongst pieces of code or programs in the policy (which we introduce later in this chapter).

Example 5. Consider a C source file s, labelled with information i_s. When compiling such source code and linking it with external libraries $l_1, \ldots, l_n$, which files are respectively labelled with information $i_1, \ldots, i_n$, the resulting binary program file f is tainted with $S = \{i_s, i_1, \ldots, i_n\} \in \wp(\mathcal{I})$, *i.e.* $itag(f) = S$.

3.3 Types of containers

At the operating system level, containers of information do not all behave the same. We found that several kinds of containers have different properties. The first distinction we make concerns subjects and objects. This notion is similar to the one used in access control models, where each subject is able to perform actions on a set of objects. We use the terms *active* containers to refer to subjects, and *passive* containers to refer to objects. We also make a distinction between containers regarding their storage locations. We consider containers stored in memory as *volatile* containers, as such containers would not survive power failure. Furthermore, even when no power failure occurs, the lifetime of such containers is limited: most of them are destroyed after a given time of execution, *e.g.* a socket is destroyed once a connection expires, a bunch of memory pages is freed once a process calls `free`[6] *etc.*. Therefore, we define:

- The set of volatile containers (objects) as C_V.
- The set of persistent containers (objects) as $\mathcal{C}_P$.
- The set of processes (subjects) as $\mathcal{C}_\Pi$.

The set of all containers is defined as:

$$\mathcal{C} = \mathcal{C}_V \cup \mathcal{C}_P \cup \mathcal{C}_\Pi$$

It should be emphasized that users are not considered as containers in our extended model. Processes are the only active agents of the system and thus we consider those as the only active containers. Processes act as subjects, running code doing operations on behalf of users, towards objects being either volatile (*e.g.* sockets) or persistent (*e.g.* files) containers. Therefore, confining users as well as programs is done at the level of processes, and only the three previously introduced types of containers exist in our model: volatile, persistent and processes.

[4] $\wp(A)$ (powerset) denotes the set of all the subsets of A.

[5] Empirically, processes are containers running the code of programs. The set of all programs is Π, therefore the set of all processes is noted C_Π.

[6] In C programming, memory is allocated by calling the C library function `malloc` and released by calling the function `free`.

3.4 Supervision of processes

As mentioned previously, we follow Denning's assumption that "processes are the active agents responsible for all information flows". Therefore, tainting rules apply to operations made by processes and involving potential[7] information flows. As we distinguish (passive) data from (active) code in the meta-information used in *tags*, different tainting rules are applied, depending on the access mode and the kind of meta-information involved.

Definition 6. For any container c,

- the function $itag : \mathcal{C} \rightarrow \wp(\mathcal{I} \cup \mathcal{X})$ returns the information tag of c.
- the function $ptag : \mathcal{C} \rightarrow \wp(\wp(\mathcal{I} \cup \mathcal{X}))$ returns the policy tag of c.

In the following, we represent the operating system as a state-transition system:

$$\sigma_i \rightarrow_{\tau_i} \sigma_{i+1}$$

We note $\sigma_0, \sigma_1, \ldots, \sigma_n$ the states of the system, and $\tau_0, \tau_1, \ldots, \tau_n$ the transitions between these states. We consider the `read`, `write`, and `exec` operations made by processes to be transitions between states of the containers. Therefore, each information flow is represented as a transition between two states i and $i+1$, respectively referring to the state of information *before* and *after* the information flow occurred.

3.4.1 Keeping tracks of running code

Before going into the details of tainting rules, let us clarify how this distinction between code and data in meta-information affects the meaning of information tags. When describing the VTT model in Chapter 2, we stated that *information tags* indicate the origin of content in containers. This remains true when considering elements of $\mathcal{I}$, describing *passive* information. However, elements of the new set $\mathcal{X}$ do not have the same meaning. Instead, such elements keep tracks of active code involved in information flows. In other words, the combination of elements of $\mathcal{I}$ and $\mathcal{X}$ in *information tags* has a dual meaning, stating which couples ⟨information, code⟩ are involved in information flows. It also depends on the kind of container:

- Any element $a \in X$ in the information tag of a processes defines that the process potentially runs this code.
- Any element $a \in X$ in the information tag of any passive container indicates a process running such code wrote information in the container.

This allows us to express additional properties in the information flow policy. We will come back to this later in this chapter.

3.4.2 Write access

When a process p accesses a container c in `write` access, we distinguish two situations: either the process *overwrites* the existing content, or it *appends* new information to the container. In the first case (overwrite), we propagate the *information tag* of the process as-is towards the container.

$$itag(c)_{i+1} = itag(p)_i$$

In the second case (append), the union of both the container and the process's *information tags* is used as the new tag for the container:

$$itag(c)_{i+1} = itag(p)_i \cup itag(c)_i$$

In any case, the new information tag of the container is tainted by both elements of $\mathcal{I}$, *i.e.* information that the process was holding at the time of the access, and elements of $\mathcal{X}$, *i.e.* which code the process was running at this time. This property allows to keep tracks of which processes write information to containers, and to express policy rules based on it, as presented later in this chapter.

[7] Recall that our analysis takes a maximum estimation of the possible content of containers into consideration.

3.4.3 Execution

Recall the *Run* relation from previous Section 3.2 of this chapter. This relation characterizes the execution of code or programs.

Definition 7. We extend the *Run* relation from Definition 4 as follows, so as to work with sets of elements rather than individual elements:

$$Run : \wp(\mathcal{I}) \rightarrow \wp(\mathcal{X})$$

$$Run(A) = \{Run(a) | a \in A\}$$

When a new process is created as the result of the execution of some code, its *information tag* is initialized with the image of the information that was executed, through the relation *Run*. Therefore, for any process p running code stored in a persistent container c, the *information tag* of the new process is initialized as follows:

$$itag(p)_{i+1} = Run(itag(c)_i \backslash \mathcal{X})$$

Elements of $\mathcal{X}$ in a running process give information related to the code that is currently running. These meta-information also taint the containers where processes write information, as described previously. Therefore, upon execution of content store in a container c, we discard elements of $\mathcal{X}$ from the *information tag* of c: we do not want taint the new process with previous *writers* of c (*i.e.*, pieces of code being executed by previous processes which wrote information to c).

3.4.4 Read access

When a process p accesses a container c in `read` access, it is tainted by the information tag of c, as follows:

$$itag(p)_{i+1} = itag(p)_i \cup (itag(c)_i \backslash \mathcal{X})$$

We discard elements of $\mathcal{X}$ for the very same reasons described previously for the case of *execution*.

3.4.5 Summary of tainting rules

Operation	$i \in \mathcal{I}$	$x \in X$
`Read`	taint	discard
`Write`	taint	taint
`Execute`	taint with $x = Run(i)$	discard

Figure 3.1: Tainting rules

As shown on Figure 3.1, we apply different tainting rules, depending on whether processes `read`, `write` or `execute` content. In this figure, *taint* means that the destination process or container gets tainted by the meta-information. *Discard* means the destination process or container does not get tainted by the meta-information. The latter only applies to elements of $\mathcal{X}$, *i.e.* meta-data attached to active code being executed.

3.5 Extended information flow policy

Before going into further details about how the (information flow) policy is attached to containers, let us define the policy itself. In our model, the policy can be expressed separately for users, programs and persistent containers. It should be emphasized that *volatile* containers do not have a

policy because these directly depend on the processes creating them and acting on them. Checking their content against the policy is done every time the acting process performs an operation tainting its own *information tag*.

For any given system, let $\mathcal{U}$ be the set of all users, $\mathcal{C}_P$ the set of all persistent containers and Π the set of all programs (*i.e.* executable code).

The information flow policy is a triplet: $\mathbb{P} = (\mathbb{P}_{\mathcal{C}_P}, \mathbb{P}_{\mathcal{U}}, \mathbb{P}_{\Pi})$ where: $\mathbb{P}_{\mathcal{C}_P}$ is the policy attached to persistent containers, $\mathbb{P}_{\mathcal{U}}$ is the policy attached to users, and $\mathbb{P}_{\Pi}$ is the policy attached to the executable code of programs.

- $\mathbb{P}_{\mathcal{C}_P} \subseteq \mathcal{C}_P \times \wp(\mathcal{I} \cup \mathcal{X})$.
  For any persistent container c protected by the policy, $\mathbb{P}_{\mathcal{C}_P}$ defines one or several sets $S = \{a \cup b\}, a \in \wp(\mathcal{I})$ and $b \in \wp(\mathcal{X})$ where:
  1. Any subset of a may legally flow into c.
  2. Applications or programs running any subset of b as their code are allowed to write information into c.
- $\mathbb{P}_{\mathcal{U}} \subseteq \mathcal{U} \times \wp(\mathcal{I} \cup \mathcal{X})$.
  For any user u that is supervised by the policy, $\mathbb{P}_{\mathcal{U}}$ defines one or several sets $S = \{a \cup b\}, a \in \wp(\mathcal{I})$ and $b \in \wp(\mathcal{X})$ where:
  1. Processes on behalf of u are allowed to access any subset of a.
  2. Processes on behalf of u are allowed to *execute* any subset of b.
- $\mathbb{P}_{\Pi} \subseteq \Pi \times \wp(\mathcal{I} \cup \mathcal{X})$.
  For any executable information π that is supervised by the policy, $\mathbb{P}_{\Pi}$ defines one or several sets $S = \{a \cup b\}, a \in \wp(\mathcal{I})$ and $b \in \wp(\mathcal{X})$ where:
  1. Processes running π as their code are allowed to read any subset of a.
  2. Processes running π as code are allowed to execute any subset of b.

The (information flow) policy is attached permanently to persistent containers and to users, and dynamically to processes as these are created, in their *policy tags*.

Definition 8. We define the relation *maycontain* as follows:

$$\forall x \in \{\mathcal{C}_P, \mathcal{U}, \Pi\}, (c, p) \in \mathbb{P}_x \Leftrightarrow c \; maycontain \; p$$

where $c \in x$ and $p \subseteq \wp(\mathcal{I} \cup \mathcal{X})$

Therefore, for any container $c \in \mathcal{C}$

$$ptag(c) = \{p | c \; maycontain \; p\}$$

3.5.1 Constrained and unconstrained containers

Unconstrained containers have no policy attached to them, *i.e.,* their policy tags are empty, whereas constrained containers have a policy tag defining their legal content. For any $c \in \mathcal{C}$,

- If c is unconstrained, then $ptag(c) = \varnothing$.
- If c is constrained, then $ptag(c) \neq \varnothing$.
- If c is constrained, and must remain empty, then $ptag(c) = \{\varnothing\} = \bot$.

3.5.2 Persistent policy

Tags are permanently attached to persistent containers when the policy is defined and applied. These are distributed in all persistent containers in the system. We qualify such a policy as *permanent* because it will remain until a new policy is defined and distributed over the system again, replacing the policy in place. Killing processes, rebooting the system or power failures will not alter such policy tags.

We attach two tags describing a policy to each persistent container: the first one describes the legal content into the container, and corresponds to a set of rules included in $\mathbb{P}_{\mathcal{C}_P}$. (The set of all of the policy tags of persistent containers is equal to $\mathbb{P}_{\mathcal{C}_P}$).

The second one defines the policy attached to the potential executable content of the container (program or code such as shared libraries). We call it *execute policy tag*, as it is used only when the content is *executed*. The set of all of the *execute policy tags* of persistent containers is equal to $\mathbb{P}_{\mathcal{C}_\Pi}$. We call this tag the *execute policy tag* of the container.

3.5.3 Initialization

At the time when we set up the (information flow) policy, *i.e.* before we start to track information flows, we attach *information tags*, *policy tags*, and *execute policy tags* to the persistent containers we wish to track. Recall that processes do not exist at this stage, and are dynamically tagged as they are created when the system is running.

Initially, information tags are initialized to unique meta-information describing the initial content of the container. This initial information is considered as being *atomic*[8]. Therefore, the minimal policy tag of any container allows at least this initial information.

Definition 9. For any persistent container c, we note $ptag_0(c)$ its initial *policy tag*. It is defined as the set of elements of the policy regarding this container, that we note $c.policy$[9]

$$\forall c \in \mathcal{C}_P, ptag_0(c) = c.policy$$

with:

$$c.policy = \{S \in \wp(\mathcal{I} \cup \mathcal{X}) | (S, c) \in \mathbb{P}_{\mathcal{C}_P}\}$$

Definition 10. For any persistent container c eventually containing executable information, we note $xptag_0(c)$ its initial *execute policy tag*. It is defined as the set of elements in the policy regarding the execution of its content π (executable code or program).

$$\forall c \in \mathcal{C}_P, xptag_0(c) = \pi.policy$$

with:

$$\pi.policy = \{S \in \wp(\mathcal{I} \cup \mathcal{X}) | (S, \pi) \in \mathbb{P}_\Pi\}$$

When no initial executable content exist in the container, we do not attach an *execute policy tag* to it.

3.5.4 User policy

As opposed to persistent containers, where the policy is distributed in each container, the policy attached to each user is globally defined in the system (*e.g.* in a hash table). The policy for each user is defined as a set of rules included in $\mathbb{P}_\mathcal{U}$. Figure 3.2 illustrates the user's policy for a system with N users.

[8] Atomic information are the smallest pieces of information that we are able to distinguish in the system.

[9] The theoretical notation *c.policy* refers to the set of rules of the policy restraining the container c. It differs from the notation $ptag(c)$, which denotes the *policy tag* attached to c, *i.e.* $ptag(c)$ contains *p.policy*.

uid_1	$set_1 \in \wp(\wp(\mathcal{I} \cup \mathcal{X}))$
uid_2	$set_2 \in \wp(\wp(\mathcal{I} \cup \mathcal{X}))$
uid_N	$set_N \in \wp(\wp(\mathcal{I} \cup \mathcal{X}))$

Figure 3.2: User policy

3.5.5 Processes

Because processes are dynamically created by the operating system upon execution of a program, *tags* cannot directly be attached to processes before these actually exist. Instead, this is done at runtime, at the time of execution. The policy for a process depends on the user on behalf of whom it performs actions, as well as the program or code being run.

The policy tag of a process determines which are the legal information flows the process can perform, given the context $\langle user, program \rangle$. The policy regarding the running program is stored in the *execute policy tag* of the persistent container storing its code on disk. This policy is used along with the policy attached to the current user, in order to determine the policy tag of the process.

Definition 11. The policy restraining a process p running a program π on behalf of user u is dynamically computed upon execution, as follows:

$$p.policy = u.policy \sqcap \pi.policy$$

where:

$$A \sqcap B = \{a \cap b | a \in A, b \in B\} \backslash \{\varnothing\}$$

Formally, $A \sqcap B$ denotes the intersection of all the common sets of A and B. After this, the *policy tag* of the new process is initialized to *p.policy*.

3.6 Legality of information flows

The legality of information flows remains the same as in the VTT model. Recall Definition 3 from Chapter 2. Intuitively, an information flow is *legal* if and only if the *information tag* of the destination container, after the information flow occurred, is included in one of the sets of its *policy tag*.

The legality of information flows is verified each time an *information tag* is updated, *i.e,.* after each information flow towards a container.

3.6.1 Initialization of processes

In the case of the execution of programs, the state of the resulting new process must be verified, to check whether the execution is legal. Figure 3.3 summarizes the creation and initialization of the *tags* attached to processes upon execution of code (or binary programs in this figure, though this applies as well to any other form of execution, such as executable memory mappings, see Chapter 4 for more details). When a new process is created, its *policy tag* and *information tag* are initialized according to the rules defined in the previous sections of this chapter:

- The policy tag of the process is set according to the policy for the current user, as well as the policy for the program being executed.
- The information tag of the process is set at runtime, as the image of the meta-data of the executed code through the relation *Run*. Such code is stored in a persistent container, which *execute policy tag* contains the appropriate policy to restrict its execution.

After the execution of a process p, we ensure that $Legal(itag(p), ptag(p))$ stands, *i.e.*,

$$\exists S \in ptag(p) | itag(p) \subseteq S$$

In Figure 3.3, *ptag*, *itag*, and *xptag* refer to the policy tags, information tag and execute policy tag of the containers, respectively.

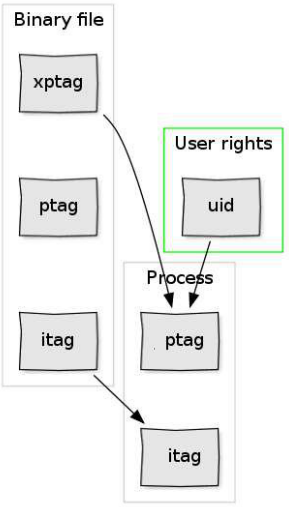

Figure 3.3: Execution of a binary program

3.7 Lattice

The following demonstration shows that the policy in our model is based on a lattice. In order to demonstrate this property, we need to introduce the following definitions.

Definition 12. Let SC be the set of all the security classes. A security class is a subset of the policy such that for any $s \in SC, s \in \wp(\wp(\mathcal{I} \cup \mathcal{X}))$

Security classes are subsets of the security policy attached to containers of information. In practice, those are either *policy tags* or subsets of the policy attached to users.

Definition 13. We introduce the relation $\sqsubseteq$ such that for any $C_1, C_2 \in SC, C1 \sqsubseteq C2 \Leftrightarrow \forall A \in C1, \exists B \in C2 : A \subseteq B$

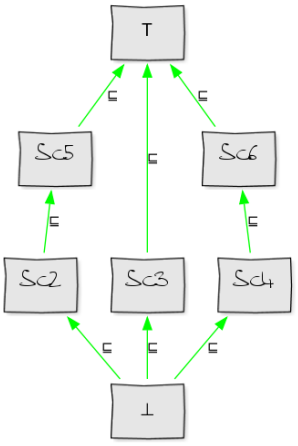

Figure 3.4: Lattice

From the previous definitions, we can establish that $\langle SC, \subseteq \rangle$ forms a partially ordered set:

- $\sqsubseteq$ is reflexive: $\forall C \in SC, C \sqsubseteq C$
- $\sqsubseteq$ is transitive: $\forall C_1, C_2, C_3 \in SC, C_1 \sqsubseteq C_2 \wedge C_2 \sqsubseteq C_3 \Rightarrow C_1 \sqsubseteq C_3$
- $\sqsubseteq$ is antisymmetric: $\forall C_1, C_2 \in SC, C_1 \sqsubseteq C_2 \wedge C_2 \sqsubseteq C_1 \Rightarrow C_1 = C_2$

Under the following assumptions, we can establish that $\langle SC, \sqsubseteq, \top, \bot \rangle$ forms a universal bounded lattice:

- SC is finite.
- SC has a lower bound $\bot = \{\{\}\}$ such that $\forall C \in SC, \bot \sqsubseteq C$.
- SC has a greatest bound $\top = \wp(\wp(\mathcal{I} \cup \mathcal{X}))$ such that $\forall C \in SC, C \sqsubseteq \top$.

This shows that the security classes of the policy in our model can be represented in a lattice. Our model can be represented in Denning's mathematical framework "suitable for formulating the requirements of secure information flow amongst security classes" [7].

3.8 Derivation from a MAC policy

In this section, we present an algorithm to derive an information flow policy, usable in our IDS, from a MAC policy as used by AppArmor, a LSM module presented in Chapter 1. This work has been published along with our theoretical model in the proceedings of the IEEE International Conference on Communications in 2011 [65].

AppArmor [54], introduced in Chapter 1, is a Linux security module enforcing a Mandatory Access Control (MAC) policy. In the following paragraphs, we provide a solution for deriving an AppArmor MAC policy into an information flow policy, usable in our model, and restricting the execution of code or programs ($\mathbb{P}_\Pi$). Such a policy does not specify rules based on users, and thus the subset of the policy concerning users ($\mathbb{P}_\mathcal{U}$) remains empty. As we monitor information flows, we discard pure access control rules which are unrelated to any possible information flow[10]. In AppArmor, the policy is composed of so-called *profiles*, where each profile describes a set of rules specific to an application (program). In order to derive a policy from a set of AppArmor profiles, and which we can use as an information flow policy for our IDS, we proceed as follows: for each statement of each AppArmor profile, we check whether such a statement is related to a potential information flow, and transform it into a corresponding statement in our model if it does. The ability to derive an information flow policy from such a wide spread format[11] leads to two major advantages. First, the specification of the policy for a given program or application can be a burden in some cases, as a lot of operations and information may be accessed by the application. Secondly, it can be very useful to use a common policy specification when comparing different models together.

3.8.1 AppArmor profiles

In an AppArmor profile, the permission granted to a program π over a object o can be one of the following: `(r,w,l,m,ix,px,Px,ux,Ux)`. These permissions are listed in Figure 3.5.

AppArmor profiles also constrain access to network resources and POSIX capabilities. However, these are pure access control rules and thus these are not taken into account in our model. Instead, possible information flows related to access to information are captured.

Definition 14. An AppArmor policy $\mathbb{P}$ is a set of profiles. A profile $p \in \mathbb{P}$ is a set of rules of the form (o, α) where o is an object and α is a permission. All these rules confine a given program $\pi \in \Pi$. We formally define such a profile as follows:

$$p = (\pi, \{(o_1, \alpha_1), \ldots, (o_n, \alpha_n)\})$$

[10] Creating a file does not cause any direct information flow as long as nothing is written in it. However, reading or writing information from/to a file does. We do only rely on such rules possibly responsible for information flows.

[11] AppArmor is used by default on some Linux distributions.

`r`	**read** (executing also needs this permission)
`w`	**write**
`a`	**append**
`l`	**link mode**: mediates access to symlinks and hardlinks
`m`	**allow executables mapping**: mmap
`ix`	**inherit execute mode**: The resource inherits the current profile, even if a profile already exists for this resource. AppArmor normally makes a transition to the profile of the newly executed program on `execve`. However, it is sometimes wanted to keep the current profile's permission while executing the new program (so as to avoid loosing permissions of the current profile, or gaining new permissions from the target's profile).
`px`	**discrete profile execute mode**: if no profile is defined for the resource, execution is denied. This requires a profile for the executed program and forces a transition to the new profile upon execution.
`Px`	**discrete profile execute mode/scrub the environment**: same as px but scrubs the environment before execution. It will tell glibc to clean the environment before executing the resource, by clearing some environment variables which may be used to modify the behavior of programs. It helps protect against e.g. LD_PRELOAD abuse. This is done by using the kernel's unsafe exec routines (otherwise, the kernel only scrubs the kernel environment in specific situations, such as the execution of setuid/setgid binaries).
`ux`	**unconstrained execute mode**: no profile is needed to execute the target.
`Ux`	**unconstrained/scrub the environment**: same as ux but scrubs the environment (see above).

Figure 3.5: AppArmor access modes

In order to compute an information flow policy which we can use in our model, from an AppArmor policy, we proceed as follows:

- For each AppArmor profile, we attach an *information tag* to each object (persistent containers) whose accesses are restricted by the profile, and we initialize it with a unique identifier.
- For each rule of a profile, we infer legal information flows towards the involved objects, and set the policy tags to these objects using the algorithm described in Figure 3.6.

3.8.2 Algorithm

The algorithm presented in Figure 3.6 transforms an AppArmor policy (a set of profiles) into an expression of an information flow policy (set of policy tags on containers). Let P be the set of all the AppArmor profiles in the policy. For any profile $p \in P$, $p.container$ is the container associated to the binary program constrained by p, $p.canread()$ is the list of files on which a $read\_like$ access is authorized, $p.canexec()$ is the list of executable files allowed to be executed, and $p.canwrite()$ is the list of paths where it is allowed to write. TOP represents the set of all atomic information tags

in the system (it corresponds to $\top$), $inherit(p) : bool$ returns *true* if the profile p inherits from its parent's profile and *false* otherwise. $unconstrained(p) : bool$ returns *true* if the associated program (subject) is unconstrained and *false* if not. $Run(I)$ is defined in Section 3.2.

```
function tag(P)
for each p in P ; do
    class = Run(itag(p.container))
    if unconstrained(p)
        data = TOP
        code = TOP
    else
        for r in p.canread() ; do
            data += itag(r)
        end
        for x in p.canexec() ; do
            code += Run(x)
        end
    end
    xptag(p.container) = data + code
    for w in p.canwrite() ; do
        w.ptag += data + class
    end
end
end
```

Figure 3.6: Derivation algorithm.

3.8.3 Examples

The following two examples respectively show how we can derive an information flow policy from a simple AppArmor profile, and how intrusions are detected by our model when using such a derived policy as well as how it compares to access control with respect to the detection of illegal information flows (we consider AppArmor being setup in *permissive mode*). Here, the security is centered on programs, with no user-related policy rules.

```
{/usr/bin/apache,
    {(/etc/apache2.conf, w), (/etc/apache2.conf, r),
    (/www/index.php,r),  (/usr/bin/ftpd, px)}
}
{/usr/bin/ftpd,
    {(/etc/ftpd.conf,w),  (/etc/ftpd.conf,r),
    (/home/ftpd/data,w)}
}
```

Figure 3.7: Example profile for derivation

Example 6. Consider the AppArmor policy example shown in Figure 3.7, where two programs are confined : *apache* and *ftpd*. Both own files that the other is not allowed to read. Using the algorithm in Figure 3.6, we can derive an information flow policy and compute its expression on the tag system. This leads to the information flow policy shown in Figure 3.8.

Example 7. The following execution sequence takes place, as presented in Figure 3.1. The apache process first reads its configuration file `/etc/apache2.conf`. Then it reads and interprets

path	itag	ptag	xptag
/usr/bin/apache	$\{i_1\}$	$\{i_1\}$	$\{Run(i_1), Run(i_2), i_3, i_6\}$
/usr/bin/ftpd	$\{i_2\}$	$\{i_2\}$	$\{Run(i_2), i_4\}$
/etc/apache2.conf	$\{i_3\}$	$\{Run(i_1), i_3, i_6\}$	$\top$
/etc/ftpd.conf	$\{i_4\}$	$\{Run(i_2), i_4\}$	$\top$
/home/ftpd/data	$\{i_5\}$	$\{Run(i_2), i_4, i_5\}$	$\top$
/www/index.php	$\{i_6\}$	$\{Run(i_1), i_3, i_6\}$	$\top$

Figure 3.8: Tags derived from the policy

`/www/index.php`, containing a security flaw. Arbitrary code is injected and executed through apache. It introduces a malware in the binary code of `/usr/bin/ftpd`. In this first part of the execution, the process running apache is not expected to write into `/usr/bin/ftpd` : the policy tag of this container is not allowed to receive information by a process running apache. Furthermore, the information apache previously read (and figuring in its information tag) does not belong to the policy tag of `/usr/bin/ftpd`. In such a situation, both AppArmor (configured in *permissive* mode) and our reference monitor would report an alert.

Then, *apache* runs the modified *ftpd*. The process running *apache* is allowed to execute *ftpd* in the security policy, hence AppArmor would allow this execution. But here, the information tag of *ftpd* has been modified when the arbitrary code was written into it, and meta-information have been added to it. Those new meta-information do not figure in the policy tag of the process running *apache*, thus it is not authorized to run *ftpd* anymore, and our reference monitor would trigger an alert for illegal code execution.

3.9 Conclusion

In this chapter, we have presented a model of intrusion detection based on an information flow policy, dynamically checking that it is respected. The policy specifies which pieces of information may be combined together and which ones the containers are allowed to contain. This model offers high expressiveness since we are able to assign meta-information to any data in the system and to constrain the behavior of programs when those data are involved. The policy expresses restrictions on access to information regardless of where it is located in the system by using a tag system associating meta-information to information containers. We explain how we maintain tags when information flows occur and how we can check whether the policy is respected. A central concept of this model is the execution of programs. This model performs dynamic checking at execution time, and is able to detect executions of illegal code or illegal flows of information. Today's MAC implementations in the Linux kernel come with extensive default security policies. It is possible to set up a policy for the model we propose from an existing MAC policy. We have shown how to derive a Blare information flow policy from an AppArmor MAC policy, and we gave an example of practical use. This model and its implementation (introduced in Chapter 4) represent our first contribution. Our new model differs from existing information flow models in the literature, such as Flume [40], Asbestos [21], Histar [76] and other DIFC models, using integrity and secrecy labels for enforcing the information flow policy. Such models are similar to multi-level security and use security classes, but provide declassification mechanisms to application programmers, so as to decentralize the authority. However, in such models, code and data are similarly considered as information, and no distinction is made between the two. For instance, a process labeled with a given secrecy level may not access a piece of code that is stored in a file with a higher secrecy level. On the contrary, our model defines the legal interactions of users and applications' code with respect to each individual pieces of information, allowing to track access to information and the execution of code separately. Furthermore, existing DIFC models do not keep the history of

state	action	itag(π_1)	itag(π_2)	itag(/usr/bin/ftpd)	alert
0	$\pi_1 = exec(/usr/bin/apache)$	$Run(i_1)$		i_2	
1	(apache,/etc/apache2.conf,r)	$Run(i_1), i_3$	$\varnothing$	i_2	
2	(apache,/www/index.php,r)	$Run(i_1), i_3, i_6$	$\varnothing$	i_2	
3	(apache,/usr/bin/ftpd,w)	$Run(i_1), i_3, i_6$	$\varnothing$	$i_2, i_3, i_6, Run(i_1)$	Both AppArmor and Blare
4	(apache,/usr/bin/ftpd,x)	$Run(i_1), i_3, i_6$	$Run(i_2), Run(i_3), Run(i_6)$	$i_2, i_3, i_6, Run(i_1)$	Blare only
	$\wedge \pi_2 = exec(ftpd)$			$i_2, i_3, i_6, Run(i_1)$	
5	(ftpd,/home/ftpd/data,w)	$\varnothing$	$Run(i_2), Run(i_3), Run(i_6)$	$i_2, i_3, i_6, Run(i_1)$	

Table 3.1: Execution sequence

information flows. While this last aspect is not required when *enforcing* the security policy, it is a major advantage when tracking information flows as it provides useful information, *e.g.,* for diagnosis of attacks or malware analysis.

TaintDroid [22] is a related work using taint data in a similar manner, however it focusses on privacy issues by attaching taint information to specific pieces of information, so as to track their illegitimate use. TaintDroid does not allow for a fine-grained policy definition, and instead relies on basic non-interference mechanisms. It is specifically designed for the Android platform, and does not allow to track *off-the-shelf* applications (such as binary applications) which is one of our requirements to provide system-wide supervision.

Chapter 4

Implementation

This chapter presents our implementation, which we built from the model described previously, in Chapter 3. This implementation is the basis for all the experiments which have been conducted during this Ph.D project. It integrates in the LSM framework, with slight modifications, and makes use of kernel standard API and data structures. It was designed to provide a generic and versatile implementation ready for future improvements and changes in the underlying model, with manageable performance overhead.

We track information flows within the operating system by using a *reference monitor* (see Section 1.4.3), which we will call KBlare[1] in the reminder of this chapter. This approach is commonly used to enforce Mandatory Access Control policies in most modern operating systems, where *subjects* may (or may not) perform a set of *operations* on *objects*. We borrowed this principle from access control mechanisms in order to track information flows between such *subjects* and *objects*, however we do not enforce any policy, but rather make use of these mechanisms to observe all information flows in a dynamic fashion. We have implemented this reference monitor at the kernel level, as it has several advantages:

- We do not need to modify userspace programs.
- We can monitor a substantial amount of information flows.
- Only kernel exploits may possibly affect our IDS.

Though our model could have been implemented in other operating systems, we have chosen to implement it in the Linux kernel for several reasons: Linux is free and open-source, it has a great community of developers and is used by the industry as well as many individuals and researchers. Furthermore, with the development of SELinux [64], the kernel developers and the NSA[2] have extended the Linux kernel with a new framework, the Linux Security Modules[3] (LSM), in order to allow different security models to be implemented (see Section 1.1.2). LSM is built on top of a set of hooks, initially suited for access control but those can be diverted to implement various security models and policies [71]. Our implementation makes use of these hooks because they provide the following advantages and guarantees:

- The code has been proven to be safe, and the hooks well placed, based on static analysis, avoiding race conditions and such flaws [79].
- The LSM framework is part of the mainstream kernel and exports a (rather[4]) stable API,

[1] KBlare is the name of this reference monitor in our open-source project.

[2] National Security Agency of the United Sates of America.

[3] When the NSA introduced SELinux in 1998 [64], the Linux kernel security was based on DAC, and did not offer any generic framework for implementing other security models and policies. Such a framework was required in order to implement SELinux.

[4] At the beginnings, the API was not quite stable, which has been widely criticized by the community. The current API, however, is much more stable and it is now an easy task to back/forward port a set of patches using LSM on any kernel version since around kernel version 2.6.26.

which simplifies the task of following the latest kernel versions.

The reminder of this chapter is organized as follows. First, we present an overview of general principles used in our implementation. Then, we present common data structures available in the kernel API, and discuss about some practical considerations regarding their use in our code. After this, we discuss about the operations involved in our analysis, and their complexity in terms of algorithms. Finally, we present all the hooks that we used in order to track information flows within the kernel, and show an exhaustive list of system calls that we track.

4.1 Overview

Our implementation builds on top of the LSM framework. Such access control hooks are used by kernel MAC mechanisms, and several modules can be chosen, one at a time, to enforce a different kind of security policy. As these hooks have been thought with access control in mind, they are not always practical for information flow control, and we have been obliged to introduced a few supplementary hooks for that matter. Also, we do not make use of all the hooks available in LSM, as a lot of these are specific to access control, and are unrelated to any potential information flow. Examples of this include hooks related to the `flock()` system call, defined in `fs/locks.c`, which triggers `security_file_lock()`. Except eventual hidden channels making use of the state of file locks, no information flows are involved in such a situation. However, it is unpractical to observe hidden channels, and this is out of the scope of this work.

4.1.1 Kernel access control hooks

The Linux kernel provides mandatory access control mechanisms, but this is not the only access control implementation available. Traditionally, discretionary access control has been used for years, and it is still the case by default on many Linux distributions. Discretionary access control has precedence over mandatory access control, in such a way that if an access is denied by DAC mechanisms, the code will return without reaching MAC related hooks and functions, as shown on Figure 4.1. In other words, MAC is more restrictive than DAC, but it does not replace it.

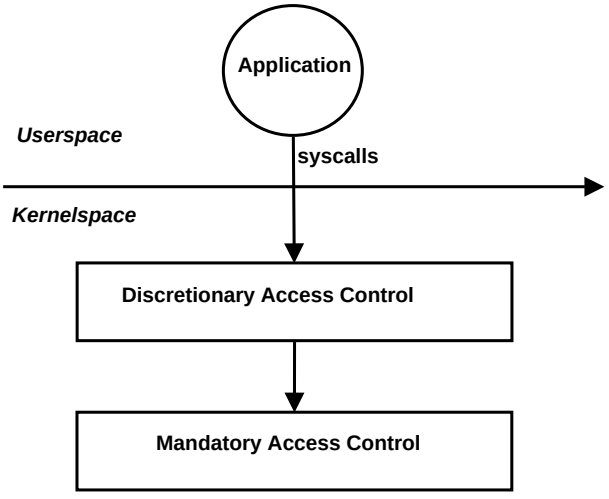

Figure 4.1: Access control hooks in the kernel

4.1.2 Tags

Recall that in our model, taint marking is performed at each object level by appending (taint) elements in sets called *information tags*, in $\wp(\mathcal{I} \cup \mathcal{X})$. *Information tags* are attached to objects, or *containers* and describe their content. Those tags are represented as ordered sets of integers in

this implementation. Positive integers represent elements of $\mathcal{I}$, whereas negative integers represent elements of $\mathcal{X}$:

- $\mathcal{I} = \mathbb{N}_*^+$
- $\mathcal{X} = \mathbb{N}_*^-$

Example 8. This is a valid information tag : {1,2,3,4,5}

The policy, determining the legality of information flows, is defined at each object level by a set of sets in $\wp(\wp(\mathcal{I} \cup \mathcal{X}))$ called *policy tag*.

Example 9. The following is a valid policy tag :
{ {1,2,3},{-4,5,6},{7,8,9} }

Tags are represented differently in system memory and on the filesystem. When representing tags in system memory, operations such as checking the legality of an information flow, or appending taint data to an existing tag must be efficient in terms of CPU and memory space. When representing tags on the filesystem, we aim to minimize the impact on input/output operations. The following section is a discussion about possible data structures to be used amongst those available in the Linux kernel.

4.1.3 Granularity

This implementation is based on Linux abstractions. The containers of information that we described in our model in Chapter 3 are operating system objects, including files, sockets, memory pages, pipes, network packets *etc.*. In order to track information flows, we attach meta-information to such OS objects. Thus, the level of granularity of our analysis is bound to the granularity of these objects. Because files are the only persistent objects of the operating system (*i.e.* stored on the filesystem, and available after reboot), we define the initial content of files as *atomic information*, as this is the finest level of granularity that it is possible to observe at the operating system abstractions level (*e.g.* we cannot distinguish information from two distinct bytes nor two distinct lines of a file from the OS perspective). In a word, *information tags* are attached to each OS object, and describe which atomic information are contained in those objects, as well as the origin of their content.

File	**Meta-information**
file 1	i_1
file 2	i_2
...	...

Figure 4.2: Atomic information in files at initialization time

Figure 4.2 is an example of how atomic meta-information can be attached to files at initialization time. Files content is described by a unique meta-information as long as their content is not altered. Whenever an information flow occurs towards a file, its *information tag* is updated so as to match the new content.

4.2 Data structures

The choice of data structures to be used is important, especially because our code runs in kernel space, where memory has a much higher cost than it has in userspace, because memory is mostly allocated in a physically contiguous manner, but also because there is a limited amount of memory available for the system kernel. It is also important to pay particular attention to evaluating the cost of all operations required by our analysis, as these occur for each information flow between

subjects and objects of the operating system, and this may considerably affect overall system performances. *Information tags* are sets of elements which could be represented as bitmaps, arrays of integers, binary trees, linked lists and other data structures. *Policy tags* are each composed of multiple sets, and may also be represented by such data structures. The following outlines our choices and compares data structures suiting our requirements. We equally refer to tags as *policy tags* or *information tags*, *i.e.* "ordered sets of meta-information".

Bitmaps

Bitmaps can be implemented with any contiguous zone in memory, such as C arrays. When representing a set with a bitmap, each bit represents a distinct element of the set, and its value in $\{0, 1\}$ represents respectively the absence or presence of this element in the set. Therefore, we need as many bits in the bitmap as there are atomic information to represent.

- Advantages : this leads to very fast logical operations using masks (logical `AND`, `OR`, *etc.*) to test the presence of individual or multiple elements at once.
- Drawbacks : it is memory hungry in situations where many files are labeled (*i.e.* the bitmap must contain as many bits as there are labeled files in the filesystem initially). Bitmaps also have a fixed size, which is not practical for our analysis, because the actual number of meta-information in *information tags* is dynamic.

Bitmaps provide good performances in the case of analyses supervising a reduced subset of the filesystem. It would be suitable if we enforced the policy instead of tracking information flows, as described in Chapter 7 (in this case, the upper bound of the size of *information tags* depends on *policy tags*), but in the present case, these are impractical when the system grows larger due to memory limitations.

Bloom filters

Bloom filters [10] are probabilistic and space-efficient data structures, and are used to represent sets of information. Testing the presence of an element in a set represented by a bloom filter can be subject to false positives, but not to false negatives. Elements can be added to the set, but not removed. Adding elements increases the probability of false positives. A bloom filter relies on a bitmap along with a variable number of hash functions. When the bloom filter is empty, all bits of the bitmap are set to 0. For any given element of the set, each hash function maps it to one positions in the bitmap, with a uniform random distribution. Adding an element to the bloom filter is done by first passing it to each hash function, and then by setting the mapped bits to 1. Testing the presence of an element is done by checking that all the mapped bits (through all of the hash functions) for a given element are set to 1. If any of the bits is not set to 1, then it is guaranteed that the element is not present in the set. Otherwise, the element may be present. Examples of use of bloom filters include symbols resolution by the dynamic linker to load shared libraries on Linux [20], where a hash table is traditionally used for the resolution. Using bloom filters to test the presence of an element before the actual lookup in the hash table leads to a dramatic increase in lookup time by filtering 80% to 90% of the unnecessary lookups. Bloom filters may be used in the same manner to avoid unnecessary lookups in *policy tags*, but using such a structure to represent *information tags* would increase the number of false positives in our intrusion detection model (*i.e.* increasing the number of intrusion alerts where no actual intrusion occurred). *Policy tags* may also directly be represented with bloom filters. Each policy tag being composed of several sets, it would require the same number of bloom filters to represent each set of the policy. However, this would lead to false negatives in our intrusion detection model, as information may be wrongly reported as present in the sets of the policy, thus allowing illegal information flows to occur.

Linked lists

The Linux kernel provides an implementation of doubly linked lists in *include/linux/list.h*. In the case of our implementation, doubly linked lists provide a scalable alternative to bitmaps, where

the size can be dynamically adjusted by inserting or removing elements without significant change in the underlying structure.

- Insertion and deletion time is in $\theta(1)$.
- Fusion sort is in $O(n)$ time if the lists are preliminary sorted (which is the case here) or $O(n.log(n))$ otherwise.

Trees

The Linux kernel provides *rbtree.h*, an implementation of the so called "Red Black Trees"[5]. Insertion, deletion and iteration cost is $O(log(n))$. According to the Linux kernel documentation, in `Documentation/rbtree`:

> *Red-black trees are a type of self-balancing binary search tree, used for storing sortable key/value data pairs. This differs from radix trees (which are used to efficiently store sparse arrays and thus use long integer indexes to insert/access/delete nodes) and hash tables (which are not kept sorted to be easily traversed in order, and must be tuned for a specific size and hash function where rbtrees scale gracefully storing arbitrary keys). Red-black trees are similar to AVL trees, but provide faster real-time bounded worst case performance for insertion and deletion (at most two rotations and three rotations, respectively, to balance the tree), with slightly slower (but still O(log n)) lookup time.*

Arrays

Arrays can also be used to represent sets. An array of int of size N for instance, noted `int[N]` in C, may be used to represent up to N distinct elements, with a memory load of $N\times$ `sizeof(int)`, where `sizeof(int)` = 32 bit (or 4 bytes) on all architectures. Allocating such an array has a very bad impact on kernel memory, as the kernel memory allocator needs a contiguous slab of $N \times 32$ bits of memory.

4.2.1 Practical considerations

The number of atomic information in a container at a given time can vary from zero to potentially (but unlikely) all the information of the filesystem (*e.g.* in the case where a single file, process or other object contains data from all the files of the filesystem). However, a lot of containers contain only one atomic information. Such containers include containers exclusively accessed read-only by all processes, and in this case their *information tags* are never tainted with any new meta-information. Usually, most containers have an asymptotic limit of possible content from various files of the filesystem. The memory overhead of *tags* depends on:

- The number σ of distinct meta-information in the system (*i.e.* how many files were initially labeled with distinct meta-information).
- The average size (length) l of *tags*, *i.e.* from how many sources does the content of containers come from.

Dynamic vs static

In the following, c_1 and c_2 are two constants respectively representing the memory space requirement per element in a static (fixed-size) structure and in a dynamic structure (*e.g.* $c_1 = 1$ in the case of a bitmap). When using fixed size data structures, such as bitmaps, the memory overhead m of *tags* is constant, and $l = \sigma$, thus:

$$m = \sigma \times c_1$$

Conversely, using dynamic data structures, such as doubly linked lists, to represent each set of the *tags*, the memory overhead m depends on the average length l of *tags*:

$$m = l \times c_2$$

[5] see `http://lwn.net/Articles/184495/` for more information about their implementation in the Linux kernel.

Example 10. Figure 4.3 is an example of a filesystem from a production server, running several services including a web server and a database. The filesystem of this server contains 66544 files. Figure 4.4 shows the maximum memory overhead per set of meta-information represented as bitmaps, arrays and doubly linked lists, considering that all the 66544 files have been labelled initially. Using bitmaps requires a constant size of 66544 for each set, where using doubly linked lists requires $((32 + 8) \times l)$[6] bits per element of the set, and arrays of integers require $32 \times l$ bits per element of the set.

```
---
# find / -print | wc -l
66544
---
```

Figure 4.3: Number of files on a Linux server.

The example on Figure 4.4 shows that, in the case of a filesystem containing 66544 files, it is preferable to use dynamic data structures when the average length of *tags* is below around a thousand of files. When containers contain information from a limited number of distinct sources, using dynamic data structures leads to a more efficient memory management. This can be generalized as follows, where l is the average length of *tags*.

$$(c_1 \times \sigma = l \times c_2) \Leftrightarrow l = \frac{\sigma \times c_1}{c_2}$$

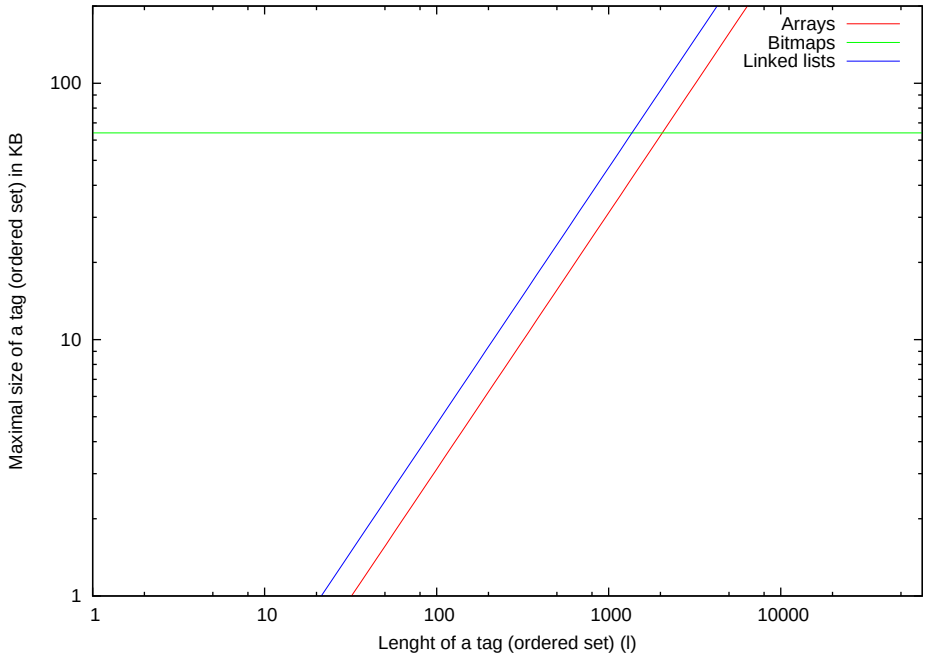

Figure 4.4: Data structures memory overhead

Memory allocation in the kernel

Another critical aspect to consider is the way the kernel handles memory. In order to minimize fragmentation due to allocation and deallocation of memory inside the kernel, the developers

[6] Linked lists of integers require at least an integer and two pointers (4 bit each) per element, `sizeof(int) = 32` and `sizeof(struct list_head) = 8`.

introduced a new mechanism called *slab allocation* [8]. This mechanism is based on the fact that initializing and destroying objects has a superior cost than allocating and freeing memory for the same objects. The so-called *slab allocator* maintains caches of the same objects types, so that the basic structure of frequently used objects is preserved between uses. When allocating memory for untypical object types, with uncommon sizes, the kernel does not directly make use of slab caches, but rather allocates chunks of contiguous memory to fit the objects. This is handled by the *buddy allocator*, which maintains caches of multiples sizes (2^k page frames each), and delivers a chunk of memory of the most appropriate size, from those available. This process involves some waste of memory: for any object o, there is a waste of $2^k * sizeof(page) - sizeof(o)$. The slab allocator itself is built on top of of the buddy allocator, so as to efficiently maintains caches of 2^k pages. In order to keep memory overhead small, it is preferable to work with small objects with common sizes, so that a slab cache is available, rather than big chunks of memory, which is more difficult for the kernel to handle, and is more likely to waste memory. In order to allocate and free contiguous chunks of memory, the kernel provides, the two functions `kmalloc()` and `kfree()`. These are implemented on top of the slab allocator, and the kernel maintains pools of various sizes for this purpose. Figure 4.5 illustrates these three layers of the memory allocation system. It is possible to directly work at any of these three levels, by invoking different functions exported by the kernel.

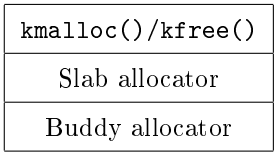

Figure 4.5: Memory allocation layers in the kernel

Statistics of the usage of slab caches are available with the `slabtop` command, as showed on Figure 4.6. Caches named `kmalloc-*` are slab caches used by `kmalloc`.

```
Active / Total Objects (% used)    : 1194073 / 1293683 (92.3%)
 Active / Total Slabs (% used)      : 44670 / 44670 (100.0%)
 Active / Total Caches (% used)     : 78 / 111 (70.3%)
 Active / Total Size (% used)       : 348175.33K / 363819.84K (95.7%)
 Minimum / Average / Maximum Object : 0.01K / 0.28K / 14.88K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
146568 146547  99%    0.66K   6107       24     97712K reiser_inode_cache
 88434  83521  94%    0.89K   5202       17     83232K ext4_inode_cache
 [...]
 73472  68244  92%    0.06K   1148       64      4592K kmalloc-64
 67116  66627  99%    0.09K   1598       42      6392K kmalloc-96
 36768  36166  98%    0.25K   2298       16      9192K kmalloc-256
 24960  20207  80%    0.03K    195      128       780K kmalloc-32
 23072  22805  98%    1.00K   1442       16     23072K kmalloc-1024
 12032  12032 100%    0.02K     47      256       188K kmalloc-16
 10592  10110  95%    0.12K    331       32      1324K kmalloc-128
  9728   8055  82%    0.01K     19      512        76K kmalloc-8
  5859   3203  54%    0.19K    279       21      1116K kmalloc-192
   848    710  83%    0.50K     53       16       424K kmalloc-512
 [...]
```

Figure 4.6: Output of the `slabtop` command.

4.3 Tags in kernel memory

Operations such as checking the legality of information flows, updating information tags *etc.* are done in kernel memory, on behalf of processes, which are "the active agents responsible for all information flows" [7].

4.3.1 Information tags

In the previous section, we have shown in paragraph 4.2.1 that using dynamic data structures minimizes memory overhead in the cases where the average number of meta-information per *tag* does not exceed a certain limit. In paragraph 4.2.1, we also made some considerations about the average size of *tags*, being either limited to a single meta-information in some cases, or bound by an asymptotic limit in other cases. Finally, we have shown that allocating small data structures with common types is handled efficiently by the kernel, by using slab caches. We therefore chose to represent *information tags*, being ordered sets of integers, in doubly linked lists, as represented on Figure 4.7. We may also have chosen to use red black trees, as both structures allow for dynamic expansion of data and make efficient use of slab caches. However, as it is shown later in this chapter, the operations we perform in our information flow analysis require iterating over all the elements of *information tags*, which makes doubly linked lists, as available in the kernel API, the most simple and efficient way to represent such ordered sets of integers. *Information tags* are represented by the following structure in the code of KBlare (defined in `security/blare/blare.h`):

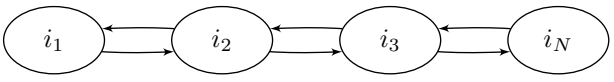

Figure 4.7: Information tags are represented as doubly linked lists

```
struct information{
    struct list_head node;
    int value;
};
```

where `node` is a structure containing information related to the list layout (`list_head` being the type for list nodes in the kernel API, containing pointers to the next and previous nodes), and `value` is an integer representing one atomic information. We decided to encode information as follows: positive values represent data (*i.e.* elements of $\mathcal{I}$), while negative values represent executed code (*i.e.* elements of $\mathcal{X}$).

4.3.2 Policy tags

Policy tags describe the legal content of containers. Contrary to *information tags*, such tags are statically defined, and thus are rarely modified, these may only be updated when changes happen in the policy. The policy attached to a container is a set of multiple ordered sets, each describing one possible combination of legal content. Each ordered set within the policy tag is represented as a balanced binary tree. It makes verifying the legality of information (against the policy) faster than it would be with a linear structure, as search operations in a binary tree are performed in $O(log_2(n))$. Binary trees of the same policy tag are linked together inside a linked list, as the process of verifying the legality of information consists in iterating over all the sets of the policy until one makes it legal. In other words, policy tags are linked lists of binary trees. Figure 4.8 shows an example of policy tag, composed of three sets, with roots $r_1, r_2, \ldots, r_N$ linked together in a doubly linked list. The following data structures are used to represent policy tags (defined in `security/blare/blare.h`):

```
struct policy{
    struct list_head list;
```

```
    struct policytree tree;
};

struct policy_tree{
    struct list_head list;
    struct rb_root *root;
    int cardinal;
};
```

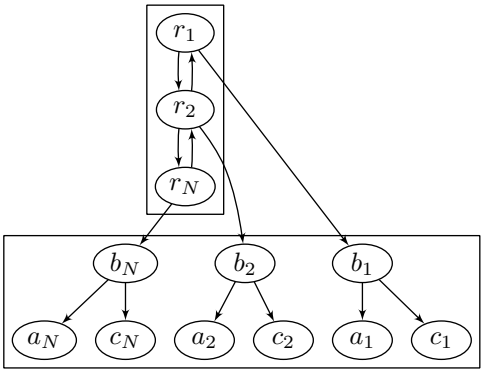

Figure 4.8: Policy tags are linked lists of binary trees

4.3.3 Execute policy tags

Recall from Chapter 3 (*Section 3.6.1*) that the policy regarding programs and executable code is distributed in the *execute policy tags* of objects. We refer to this subset of the policy as the *execute policy* and it is stored:

- On disk, in the extended attributes of files containing executable code (*e.g.* binary files and shared libraries) which we want to confine, as described later in this chapter.
- In memory, in the *execute policy tags* of processes, shared memory mappings, pipes, queues and sockets, which we will further discuss here.

The semantics differs in each case. The execute policy tags of files are used at runtime (along with the policy of users) to determine the policy tags of processes (as presented in Chapter 3). When attached to processes, *execute policy tags* are stored in kernel memory, and are updated whenever processes either execute or read some code (*e.g.,* a shared library) with an information flow policy attached to it (*i.e.* the file containing the code has an *execute policy tag*). In such a case, the *execute policy tag* of the process is *tainted* by the *execute policy tag* of this executable content: we compute a new tag containing the common set of both *execute policy tags*, as described later in this chapter, in Section 4.6.2. The aim of tainting processes with *execute policy tags* is to make sure the *execute policy* of all executable content that has been accessed is kept when new information flows occur towards other containers. The following example shows a possible issue which happens if we do not taint objects with *execute policy tags*.

Example 11. File `/home/alice/flash_plugin.bin` has the following execute policy tag : {{1,2,5},{-1,2}} Now imagine that Alice (or any program on her behalf) runs :

```
alice@alicebox:~/ cat flash_plugin.bin > .firefox/plugins
```

After running this command, the shell will fork and execute `cat`, which in turn will read `flash_plugin.bin`, and output its content to another file in `.firefox/plugins`. The new file will not have any *execute policy tag* attached to it unless we do make sure *execute policy tags* get tainted.

To overcome this issue, we need to ensure that :

- Whenever a process reads a file or other object containing executable code, we read the *execute policy tag* of this object, and append it to the *execute policy tag* of the process.
- Whenever a process writes to an object, it appends its *execute policy tag* to the *execute policy tag* of the object.

By doing so, we make sure that all objects have their execute policy tag updated when code gets copied to another object. Whenever a new subset of the *execute policy* (*i.e.* an element of $\wp(\wp(\mathcal{I} \cup \mathcal{X}))$), bound to a piece of executable information, is read or executed, it is included[7] in the *execute policy tag* of the current process. When processes write information to files or shared memory mappings, the *execute policy tags* of these objects also get tainted the same way.

4.4 Tags on disk

The persistence of a system with everything running into memory is very limited. It would also be very inefficient in terms of memory to maintain in-memory data structures for every object, especially for every file of the filesystem. In order to be able to restore the state of the system after rebooting, or to be able to free in-memory information tags of files no longer accessed by any running process, tags are stored on disk, in the extended attributes of the filesystem, in the form of `name:value` pairs, each containing up to 64 KB of binary data[8]. We store values in the security namespace (`security.*`), as used by the other LSM modules.

- *Information tags* use one field of the extended attributes: `security.blare.info`
- *Policy tags* and *execute policy tags* use several fields (one *key:value* pair for each ordered set of the policy). For a *policy tag* with N subsets, fields names are:

  $$\texttt{security.blare.policy}\{k\}, \text{ with } 0 \leqslant k < N$$

  For an *execute policy tag*, fields names are:

  $$\texttt{security.blare.xpolicy}\{k\}, \text{ with } 0 \leqslant k < N.$$

Example 12. The policy tag { {1,2,3},{-4,5,6},{7,8,9} } of a given file, would be represented in three distinct key:value pairs:

- `security.blare.policy0`
- `security.blare.policy1`
- `security.blare.policy2`

4.4.1 Serialization

Serialization is the process of converting data structures from an in-memory format, into a format that can be stored, or transmitted over a network connection, in such a way that it can later be restored back to its original live form, by an operation called *unserialization*. We need a serialization mechanism in our implementation, in order to be able to store live tags into the extended attributes of the filesystem, and to restore live tags back into memory when processes access information stored into files not currently in use. The extended attributes are represented on disk as flat and contiguous sets of bytes. Such a representation requires an intermediate structure that is contiguous in memory, so that we can dump it into a *key:value* pair (*i.e.* we cannot write non contiguous data structures in the extended attributes). We chose to use ordered arrays of integers for this purpose. On access to files, meta-information into tags are converted from their disk representation to their memory representation, and vice-versa.

[7] The result is the union of the two sets of sets.

[8] According to the manpage of `attr`, extended attributes on XFS filesystem objects on Linux.

- On read accesses, meta-information is read from the extended attributes and converted into a live representation (tree or list).
- On write accesses, meta-information is written to the extended attribute and thus converted into a linear representation (continuous memory region).

Example 13. `int array[6]` allocates `6 * sizeof(int)` in a contiguous region of memory.

In kernel code, operations that we can perform on inodes, such as operations on the extended attributes, are associated to each inode structure. The following two operations are used in our implementation, and are available after filesystem initialization (the kernel would return `-EOPNOTSUP`[9] before this stage, or when extended attributes are unavailable on the filesystem in use).

```
inode->i_op->getxattr(dentry, name, buffer, size)
inode->i_op->setxattr(dentry, name, buffer, size)
```

4.5 Users policy

Recall from Chapter 3 that in our model, the information flow policy is composed of three distinct subsets, $\mathbb{P}_\Pi, \mathbb{P}_\mathcal{U}$ and $\mathbb{P}_{\mathcal{C}_P}$, respectively expressing the policy regarding executable code, the policy regarding users, and the policy regarding containers. The following gives implementation details about $\mathbb{P}_\mathcal{U}$.

4.5.1 On disk

A user policy defines what a user (or `uid`) is allowed to do. In practice, it is used to determine which subsets of information a process on behalf of a given user is allowed to access. For each `uid`, a user policy can be defined, and is similar to the policy tag of containers in the sense that it is a set of ordered sets.

Example 14. The following is a valid user policy : { {1,2,3},{-4,5,6},{7,8,9} }.

Where the policy tags can be stored in the extended attributes of each file, user roles need to be centrally defined somewhere on the filesystem. The policy for each user (`uid`) can be defined and stored, from userspace, in the extended attributes of a file `/etc/blare/uid`. This ensures that the users policy is stored in a persistent fashion, and it allows us to restore it at boot time.

4.5.2 In memory

The policy for each user is stored in a linked list of binary trees, the same way *policy tags* and *execute policy tags* are represented in memory. It is used at runtime along with *execute policy tags* to compute the *policy tags* of processes (as presented in Chapter 3).

4.5.3 Communication between userspace and kernelspace

The kernel should not directly read the special file storing users policy in the filesystem (`/etc/blare/uid`) because the location of such file is a *policy* and thus should not be defined within kernel code. Instead, we use the `securityfs` interface (mounted as `/sys/kernel/security`), which exports a pseudo-filesystem available from userspace, to load users policy at boot time. For each user, a special file is created in the pseudo-filesystem, thus allowing the system administrator to load the policy (set of sets) of this user. Each set of the policy that is loaded this way is copied into kernel memory, into a policy tag attached to the appropriate user id. Each special file is named after the `uid` of the corresponding user, and is created in `/sys/kernel/security/blare/users`. When a running process on behalf of a user runs the `exec` system call, it checks whether a policy has previously been loaded in kernel memory for this user, and uses it along with the *execute policy tag* of the executable file.

[9] Return code standing for "operation not supported".

4.6 Operations and complexity

4.6.1 Updates on information tags

Information tags are updated when information flows occur (the destination container's information tag is updated). If both the source and destination containers are in memory (sockets, processes, IPC, ...), the involved operation is the fusion of two linked lists, which complexity is $O(n + m)$ for two lists of respective sizes n and m. If one of the containers is a file, a conversion from/to a linear structure is needed (see Section 4.4.1).

- On read operations from files, the extended attributes are dumped in a memory buffer (of type `int*`). We iterate over the resulting array, and store each array value in the (in-memory) information tag (linked list) of the current process performing the read operation. See `itag_insert` from `security/blare/itag.c`.
- On write operations, we overwrite the information tag of the file with the information tag of the current process. We iterate over the linked list, and create an array of integer from it, so as to write it into the extended attributes. See `blare_write_info` from `security/blare/itag.c`.
- On append operations, we first read the information tag of the file, append it together with the information tag (linked list) of the current process into an array of `int` and write the new array to the extended attributes of the file.

A process P reading information from a container C has its *information tag* updated as follows:

$$itag(P)_{i+1} = itag(P)_i \cup itag(C)_i$$

where $i + 1$ refers to the state of the tag *after* the information flow occurred, and i refers to the state of the tag *before* the information flow occurred.

4.6.2 Updates on execute policy tags

Execute policy tags of processes are also updated dynamically, whenever an information flow occurs, potentially involving executable code confined by the *execute policy*. This operation involves the restriction of two policy tags against each other, *i.e.* the intersection of all the sets of one *execute policy tag* with all the sets of the other one. If we consider the fusion of two *execute policy tags* $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_m)$, then the complexity of this operation is in $O(n^2.m^2)$. We define this operation as $A \sqcap B$, as presented in Definition 11 previously in Chapter 3. It is implemented by the pseudo-code described in Figure 4.9.

4.6.3 Legality check

To verify the legality of an information flow between two containers, we check that the **information tag** of the destination container (C_{dest}) is **legal** with respect to its **policy tag**:

$$\exists S \in ptag(C_{dest}) | itag(C_{dest}) \subseteq S$$

The legality check is performed by iterating over the **information tag** of C_{dest}, checking that all of its elements belong to its **policy tag**. This is a linear operation with respect to the size of the information tag. Its complexity is in $O(k \times \ell \times log_2(n))$, where k is the length of the involved *information tag*, l is the number of subsets of the policy and n is the maximum size of the sets of the policy.

4.7 System calls and hooks

System calls are the interface between applications and the kernel. A lot of operations such as opening files, creating shared memory mappings or executing programs involve system calls, most

```
new := {}
size := 0
for (i := 1 to n) do
    for (j := 1 to m) do
        c:= a_i ∩ b_j
            for (k := 1 to size) do
                if (c ⊆ new[k]) then
                    break
            done
            if (k = size+1) then
                new := new ∪ c
                size++
            fi
    done
done
```

Figure 4.9: Execute policy tags intersection algorithm

often indirectly by calling wrapper functions from the C library[10] rather than directly invoking the underlying system calls. It is necessary to track system calls in order to track information flows between processes. The LSM framework provides hooks for tracking system calls involving access to information. In Appendix A, we provide a detailed list of the system calls in the Linux kernel version 3.2, where system calls which may lead to information flows are identified. In this section, we show how our reference monitor uses LSM hooks and which system calls correspond to it. All the hooks that we use are defined in `security/blare/lsm_hooks.c`. The LSM framework made changes to the structures used in kernel space to represent kernel objects, including file descriptors, inodes and processes credentials, by adding an opaque security field of type `void*`, that the LSM modules can use to store their own security attributes [71]. Furthermore, processes *credentials*[11] have been extended to support concurrent access, and now have a supplementary `void* security` field to store opaque structures. The credentials (including the security field) are protected by Read Copy Update (RCU) mechanisms [47].

Special structures

As previously described in our theoretical model in Chapter 3, containers of information are separated into three classes: volatile objects, persistent objects (*i.e.* backed on the file system), and processes. The `blare_tags` structure is used for all volatile objects. It is defined as follows:

```
struct blare_tags{
    struct list_head *info;
    /* Used by softirqs invoking rcv_skb*/
    spinlock_t info_lock;
    atomic_t refcount;
    int info_rev; //unused
    struct list_head *policy;
    struct list_head *xpolicy;
};
```

Where `info` is a pointer to an *information tag*, `policy` is a pointer to a *policy tag*, and `xpolicy` is a pointer to an *execute policy tag*. Files and processes have their own data structures, respectively `blare_file_struct` and `blare_task_struct` and are described later in this chapter.

[10] The GNU C library is the most common C library on Linux, often called glibc.

[11] Credentials are used to store various security information related to processes, and are attached to the `task_struct` structure in the `cred` field. See `documentation/credentials.txt` in the kernel source tree.

4.7.1 Fork and clone

Fork

When a process forks by calling the libc function `fork` (which in turn calls the `clone` system call with special flags), the resulting child process is an exact copy of the parent process in terms of memory, except for a couple of properties (listed in the manpage of `fork (2)`). A number of flags may affect the behavior of `fork`, by determining how the parent and the child may share system objects. Amongst those flags, `MADV_DONTFORK`, which can be set on memory mappings using the `madvise` system call, affects the semantics of possible information flows between the child and the parent. Memory mappings (described in the next subsection) are normally inherited during the fork process, unless those have been marked with this flag. Similarly, the set of open file descriptors is inherited by the child, however we track the actual access to files (through `fread` or `fwrite` system calls), so this does not affect our analysis. The same goes with open message queue descriptors, as we track actual calls to `msgget`. In all cases,

- The child's *information tag* is an exact copy of the parent's *information tag*.
- The child's *policy tag* is an exact copy of the parent's *policy tag*.
- The child's *execute policy* tag is an exact copy of the parent's *execute policy* tag.

Clone

The `clone` system call is mostly used to create threads. When the `CLONE_VM` flag is passed, the child process uses the same address space as the parent (and any call to `mmap` or `munmap` affects both processes). Otherwise, the child process has its own address space. In this later case, existing anonymous shared mappings of the parent are shared with the child. If the `CLONE_NEWIPC` flag is passed, then the child uses a new IPC namespace, and will not be able to see the objects created in the parent's namespace. If this flag is not set, the child shares the same IPC namespace, and is able to access shared memory segments through `shmat` and messages through `msgget`. In the case of shared memory segments, KBlare considers an over-estimate of the possible information flows from the time when the segment is attached with `shmat` until it is detached with `shmdt`.

Related hooks

Both clone and fork are hooked in the LSM framework (`security_task_create`) and trigger two functions in KBlare. The first one is defined as follows:

```
static int blare_task_create(unsigned long clone_flags);
```

It is not yet used in our implementation, but it gives useful information about the clone flags, which may be used in the future to track individual threads of the same process[12]. The second one is defined as follows:

```
static int blare_prepare_creds(struct cred *new, const struct cred *old, gfp_t
gfp);
```

This function is part of the RCU[13] mechanisms protecting access to the credentials of processes, and returns an exact copy of the protected structures (`blare_tags` in this case).

[12] We do not track individual threads in our current implementation, because all threads of a process share the same address space. Therefore, there is no way to dynamically track information flows between threads. In order to track threads individually, it would be necessary to ensure no information flow can possibly occur, by auditing the code, which is out of the scope of this work.

[13] RCU stands for Read Copy Update, it is a low overhead synchronization mechanism widely used in the Linux Kernel. See McKenny and Walpole's work [47] for more about RCU.

4.7.2 Memory mappings

In this section, we describe separately how KBlare deals with *shared memory mappings* (*i.e.* `mmap`) and *shared memory segments* (System V shared memory, *i.e.* `shmat`).

Mapping a file to memory

Processes have the ability to create memory mappings, by calling the `mmap` system call. Memory mappings are often used to map the content of files to memory, but they can also be used without any underlying file. In this case, it is similar to shared memory segments (as described in the next subsection).

```
void *mmap(void *addr, size_t length, int prot, int flags,
                  int fd, off_t offset);
```

Any mapped file can be unmapped by calling the `munmap` system call. Amongst the possible flags, `MAP_PRIVATE`, `MAP_ANONYMOUS` and `MAP_SHARED` change the behavior of memory mappings, and how other processes may access it.

- `MAP_PRIVATE`: the memory mapping is not visible by other processes, and changes made to the mapping are not backed to the underlying file. Conversely, changes to the underlying file may or may not affect the memory region, this behavior is unspecified by the POSIX specification. In order to keep a conservative approach, changes to mapped files should update memory mappings as well in our implementation. This particular aspect is not taken into consideration in our current implementation, and it will be fixed in the future.

- `MAP_SHARED`: updates on the memory mapping are visible to other processes mapping the same file. Content is also updated on the filesystem, but it may not actually be updated until `msync` or `munmap` is called. Note: Calling mmap with `MAP_SHARED` before a fork will make those mappings available to the child.

- `MAP_ANONYMOUS`: the memory mapping is not backed to any file. The information is kept in memory. Anonymous shared mapping are available to the child after a fork (`MAP_ANONYMOUS|MAP_SHARED`).

The protection flags also affect the way information may flow between processes and a mapping (these flags are enforced by the hardware, when possible):

- `PROT_EXEC` allows execution of the pages' content.

- `PROT_READ` allows reading the pages.

- `PROT_WRITE` allows writing to the pages.

In our implementation, *information tags* are attached to shared memory mappings, when those allow at least write access to the owner process[14]. Non shared memory mappings directly affect the process's information tag in a way that is similar to the other file system operations. Information flows between a process and a shared memory mapping are tracked until the process unmaps the file (or memory region in case no file descriptor exists).

Hooks

KBlare tracks calls to both `mmap` and `munmap`. The latter is not part of the LSM framework and had to be manually added in our kernel patch. The following hook ensures that we update the *information tags* of processes having access to mapped files or regions:

[14] Non writeable mappings occur quite frequently, *e.g.* when loading shared libraries. Such mappings are equivalent to reading the file, in terms of information flows

```
static int blare_file_mmap(struct file *file, unsigned long reqprot,
        unsigned long prot, unsigned long flags,
        unsigned long addr, unsigned long addr_only);
```

The semantics is the following, when pages are writeable:

- In the case of non anonymous shared mappings, a `blare_tags` structure is attached to the file descriptor of the mapping, in its `file->_security` field, in order to store the *information tag* of the memory mapping (we do not set any policy on the mappings, the policy verification is left to the processes, as described in the model in Chapter 3).

- In the case of anonymous shared mappings, only the child process may have access to it, unless `MADV_DONTFORK` was set. No file descriptor is available for this kind of mappings, as it is not backed to any file. *Information tags* of the parent and the child have to be kept synchronized until one of them releases the mapping. This is not supported in our current implementation. A special flag should be added to `blare_task_struct`, and set to 1 for all parents having child processes sharing memory mappings with them (this can be done in `security_task_create`).

- In the case of non anonymous non shared mappings, information is backed to the file in case of changes to the mapping.

System V shared memory

From userspace, shared memory segments are allocated by processes invoking `shmget`. Once a shared memory segment is created, processes can attach it to their address space by calling `shmat`. If the `SHM_RDONLY` flag is passed, then the calling process has read-only access to the memory segment, and otherwise it has read and write access to it. Processes detach memory segments from their address space by invoking `shmdt`. Processes attached to a memory segments can access it directly, and this is not caught by the operating system. In kernelspace, shared memory segments are represented by `struct shmid_kernel *shp`:

```
struct shmid_kernel /* private to the kernel */
{
        struct kern_ipc_perm        shm_perm;
        struct file *                shm_file;
        unsigned long                shm_nattch;
        unsigned long                shm_segsz;
        time_t                        shm_atim;
        time_t                        shm_dtim;
        time_t                        shm_ctim;
        pid_t                        shm_cprid;
        pid_t                        shm_lprid;
        struct user_struct        *mlock_user;
};
```

Each `struct shmid_kernel` embeds a `struct kern_ipc_perm`:

```
struct kern_ipc_perm{
        spinlock_t        lock;
        int                deleted;
        int                id;
        key_t                key;
        uid_t                uid;
        gid_t                gid;
        uid_t                cuid;
        gid_t                cgid;
        mode_t                mode;
        unsigned long        seq;
        void                *security;
};
```

The `security` field of `struct kern_ipc_perm` is used by KBlare to store meta-information concerning the shared memory segment. Such meta-information is stored in a `struct blare_tags`, as with other *volatile* objects.

```
struct blare_tags{
    struct list_head *info;
    /* Used by softirqs invoking rcv_skb*/
    spinlock_t info_lock;
    atomic_t refcount;
    int info_rev; //unused
    struct list_head *policy;
    struct list_head *xpolicy;
};
```

Hooks

The LSM framework provides a hook for `shmat`, but a hook for `shmdt` had to be manually added in our patch set, in order to be able to stop tracking processes after a shared memory segment is released. A process attaches a shared memory segment to his address space by invoking the `shmat()` system call. KBlare tracks this system call with the `security_shm_shmat` hook, with a callback on the following function in our security module:

```
static int blare_shm_shmat (struct shmid_kernel *shp,
                        char __user *shmaddr, int shmflg);
```

KBlare maintains a list of currently attached shared memory segments for each process (in `cred->security->shm`). For each memory segment of the list, a pointer to the tags of the memory segment (of type `struct blare_tags`), as well as the flags determining the access mode (*e.g.,* `SHM_RDONLY`) are stored in the following structure:

```
struct blare_shmptr{
    struct list_head node;
    struct blare_tags *ptr;
    int shmflg; //shmat() flags, i.e. SHM_RDONLY etc.
};
```

Processes detach a memory segment from their address space by invoking the `shmdt` system call. KBlare tracks this system call with the `security_shm_shmdt` hook, with a callback on the following function in our security module:

```
static void blare_shm_shmdt(struct shmid_kernel *shp);
```

On release of a memory segment, the following actions are performed:

- The information tag of the current process is updated with the information tag of the shared memory segment.
- The memory segment is removed from the list of attached memory segments for the current process (`cred->security->shm`).

Access to shared memory

As previously mentioned, access to attached shared memory segments is not subject to any system call and is not tracked by the operating system. Therefore, KBlare calculates an overestimation of the possible information flows between a process and its attached shared memory segments.

- When a process P reads new content and updates its information tag (`e.g.,` by reading information in a file or socket), all the shared memory segments it has attached with write (read and write) access also have their information tag updated.
- Before any process writes or appends information to a container, the information tags of all the attached shared memory segments are merged into the process's information tag.

4.7.3 Files and pipes

The most common way for processes to access information is certainly through the filesystem. Processes access files using system calls. Amongst available system calls, `fopen` and `fclose` are used to respectively open and close a file descriptor. When a file is opened, a flag called open mode is specified, and takes a value in {`a`,`w`,`r`}.

- `r` opens the file in read mode if it exists.
- `w` opens the file in write mode or create it if it does not exist. Any content in the file is overwritten (the file is truncated to zero length).
- `a` opens the file in append mode, content may be written at the end of the file, and existing content cannot be altered.
- Furthermore, `r+`, `w+` and `a+` are also valid modes. `r+` is like `r` with write access allowed, `w+` and `a+` are like `w` and `a` with read access allowed

After this, input/output access is performed by `read` and `write` or `pread` and `pwrite`. The "p" variants allow to read or write from a given offset. These system calls are responsible for information flows between processes and files, and are tracked in KBlare. Similarly, pipes can be created with the `pipe` system call, and accessed with the system calls `read` and `write`.

Hooks

In the LSM framework, access to files and inodes is verified by two distinct hooks: `security_file_permission` and `security_inode_permission`. In kernel space, file descriptors may describe regular files or pipes. Each file descriptors is linked to an underlying inode (except before filesystem initialization). When it describes a pipe, the inode has a special field `i_pipe`, which we use to distinguish it from regular files. However, inodes are also used by sockets, and other objects. As the kernel relies on inodes in many cases, hooking inode accesses results in a lot of callbacks for each process. In our implementation, we rather verify access at other levels. In the case of files, the `security_file_permission` hook triggers the following callback:

```
static int blare_file_permission (struct file *file, int mask);
```

where `file` is a pointer to the file descriptor, and `mask` is the access mask (which determines the access mode). KBlare stores its security attributes in the opaque security pointer field of the `file` structure, as introduced by the LSM framework: `file->f_security`. The security attributes we attach to files are specified as follows:

```
struct blare_file_struct{
    int *info_array;
    int info_size;
    struct policy_array **policy_arrays;
    int policy_count;
    struct policy_array **xpolicy_arrays;
    int xpolicy_count;
    struct blare_tags tags; // used for unnamed pipes
};
```

Recall from Section 4.4 that tags associated to files are stored in the extended attributes of the filesystem. Such a representation is "flat", *i.e.* it is represented as a contiguous region of memory.

- When a process reads a file, KBlare reads the extended attributes and stores it in the `info_array` data structure, and sets `info_size` to the size (number of elements) of the array. This is later converted into a "live" representation, as previously described in this chapter in Section 4.3.
- In the case of pipes, no extended attributes are used, as pipes are residing in memory, and the "live" representation is used directly by using a `blare_tags` structure.

4.7.4 Message queues

Message queues are another inter process communication mechanisms allowing processes to exchange so called messages, stored in queues. Messages have a priority, and messages with the highest priority are delivered to the receiving process first. Linux implements POSIX message queues, as well as SYSV message queues. Both use a distinct API.

SYSV message queues

As with files, and other data structures, SYSV message queues as well as their messages themselves have been modified by the LSM patches to add an opaque security field. The structure `struct msg_queue` is defined in `include/linux/msg.h` as follows:

```
struct msg_queue {
    struct kern_ipc_perm q_perm;
    time_t q_stime;         /* last msgsnd time */
    time_t q_rtime;         /* last msgrcv time */
    time_t q_ctime;         /* last change time */
    unsigned long q_cbytes;     /* current number of bytes on queue */
    unsigned long q_qnum;       /* number of messages in queue */
    unsigned long q_qbytes;     /* max number of bytes on queue */
    pid_t q_lspid;          /* pid of last msgsnd */
    pid_t q_lrpid;          /* last receive pid */

    struct list_head q_messages;
    struct list_head q_receivers;
    struct list_head q_senders;
};
```

As for shared memory structures (see Section 4.7.2), the structure for message queues embeds a `kern_ipc_perm` structure, itself having an opaque security field. However, rather than labelling the message queues, KBlare labels individual messages. Messages are defined as follows:

```
struct msg_msg {
    struct list_head m_list;
    long  m_type;
    int m_ts;           /* message text size */
    struct msg_msgseg* next;
    void *security;
    /* the actual message follows immediately */
};
```

The `security` field of this structure is used by KBlare to store its tags (in a `blare_tags` structure).

Related hooks

Two hooks are used by KBlare, and are triggered upon sending or receiving messages: `security_msg_queue_msgrcv` and `security_msg_queue_msgsnd`. Those hooks trigger the following functions of our module:

```
static int blare_msg_queue_msgrcv (struct msg_queue *msq,
    struct msg_msg *msg,  struct task_struct *target,
    long type, int mode);

static int blare_msg_queue_msgsnd (struct msg_queue *msq,
    struct msg_msg *msg, int msqflg);
```

One of the caveats with the reception of inline messages (*i.e.* fetching the first message available in the queue) is that the target process is not equal to the current process[15] in this portion of the code (the kernel runs in a different context). Whenever the target differs from the current process, we are unable to alter the credentials of the receiving process, because of the RCU protection mechanisms, forbidding a task to alter other tasks' credentials (there are good reasons[16] for this, as it would make the credentials management much more complex). The best way we found to work around this was to force the scheduler to wake up the target process:

```
    /* We cannot alter target's credentials unless it is the current process */
    if (target != current)
        wake_up_process(target);
```

POSIX message queues

Posix message queues make use of *inode* structures to pass messages. This could be tracked by using the `security_inode_permission` hook, but it is not yet supported in our implementation.

4.7.5 Networking

UNIX domain sockets, or IPC sockets, allow processes of the same host to communicate through network packets. Furthermore, *network sockets* allow processes of different[17] hosts to communicate over a network. KBlare tracks communication over *UNIX domain sockets* of type `AF_UNIX`, and *network sockets* of type `AF_INET`. After receiving messages through a socket (at state i), the new *information tag* of the process (at state $i+1$) is updated by appending the new content from the *information tag* of the socket to it.

[15] See `include/linux/security.h`.

[16] From `Documentation/security/credentials.txt`: *"[...]As previously mentioned, a task may only alter its own credentials, and may not alter those of another task. This means that it doesn't need to use any locking to alter its own credentials.[...]"*

[17] It can also be used on the same host.

$$itag(process)_{i+1} = itag(socket)_i \cup itag(process)_i$$

Similarly, when sending information through a socket, the *information tag* of the socket is updated with the *information tag* of the process in the same manner:

$$itag(socket)_{i+1} = itag(socket)_i \cup itag(process)_i$$

Kernel structures

Sockets are described in kernel space by the `socket` structure, defined as follows in `include/linux/net.h`:

```
struct socket {
    socket_state        state;

    kmemcheck_bitfield_begin(type);
    short           type;
    kmemcheck_bitfield_end(type);

    unsigned long       flags;

    struct socket_wq __rcu  *wq;

    struct file     *file;
    struct sock     *sk;
    const struct proto_ops  *ops;
};
```

The `socket` structure has a pointer to a `sock` structure, containing the network layer representation of sockets. This is a quite complex structure, and we will not fully describe it here. The `sock` structure contains a field called `sk_family`, and we use it to determine whether sockets are of type `AF_UNIX` or `AF_INET`. As other *volatile* objects, sockets are labeled by KBlare with a `blare_tags` structure, attached to their opaque security field. This field is defined in the `sock` structure as `sk_security` of type `void*`.

Related hooks

Communication over `AF_UNIX` sockets is monitored by two hooks. Sending messages is caught by `security_unix_may_send`, and receiving messages is caught by `security_socket_recv_msg`. This later hook is also triggered when receiving information through internet sockets, and KBlare treats both cases in the same place, by determining the kind of socket. Sending messages over `AF_INET` sockets is caught by `security_socket_sendmsg`.

Netlink messaging

Netlink is a communication mechanism between kernelspace and userspace. It uses BSD sockets of the `AF_NETLINK` family. It can also be used to communicate between userspace processes, even thought this is not its primary goal. Netlink messages are not supported yet in KBlare, this is left for future work. At the moment, the following stubs are defined:

```
static int blare_netlink_send (struct sock *sk,
                        struct sk_buff *skb);
static int blare_netlink_recv (struct sk_buff *skb, int cap);
```

Part III

Distributed Intrusion Detection

Chapter 5

Network Extension

In the previous chapters, we have introduced a model of intrusion detection based on taint marking techniques. It tracks information flows between objects of the operating system, and allows to detect abnormal behavior caused by intrusions on a local host. The next step towards detecting intrusions in distributed systems is to track information flows at the network level. This chapter presents a network extension to the previous model, adding further control over information with respect to outgoing traffic (the more complex case of incoming traffic is presented in Chapter 6). We have extended the previous information flow policy with a so-called *network policy*, stating how information may leave the system, restraining sockets given the current (user, code) context. Furthermore, we have developed a framework that allows users to trace how their private data is used by applications, and to monitor sensitive information that flows out over the network. This led to experiments presented in Chapter 8, and to a publication in the proceedings of the Australasian Information Security Conference (AISC) 2012 [32]. Details regarding the implementation of this network extension are presented in Chapter 7.

5.1 Overview

Most of today's personal computers rely on untrusted third party applications such as browser plugins or so called 'apps'. Many of these are closed source, which makes static analysis extremely difficult (if not impossible) in the case of native code. And even in the case of opensource applications, there is always a risk of security flaws or coding errors potentially leaking sensitive data. Dynamically detecting the leak of sensitive information is challenging given that:

- One application can exchange information with another through IPC, shared memory, etc.
- It is impractical to modify off-the-shelf applications; instead, we prefer to implement a reference monitor in the operating system kernel as a more pragmatic solution.
- The performance overhead must be small to maintain a responsive system, i.e., not affecting the user's experience and causing them to disable the security mechanisms.

As presented in Part II, we use dynamic tracking of information flows between objects of the operating system. A defining aspect of our approach is that we distinguish *data* from *containers*: data is the actual information we track, whereas containers are storage entities such as files, memory pages, etc. Sensitive data is first identified and their containers are labeled with meta-data called *tags*. As information flows between containers, tags are dynamically updated to reflect the containers' content. When it comes to protecting sensitive data against leakage by untrusted applications or via malware that exploits security flaws, existing approaches have several limitations. Individuals can use software firewalls on their internet-connected personal/portable computing devices to filter network connections without changing the security policy of the underlying operating system. However, while such mechanisms may successfully protect a host from outside threats,

they typically do not prevent the leak of information by untrusted or misconfigured applications. Deep packet inspection firewalls are able to identify data patterns in network packets, however this approach is too coarse-grained to efficiently track the presence of sensitive data in network exchanges and is thus not an effective solution to protect against sensitive data leaks. Mandatory access control tools such as AppArmor [54] and Tomoyo [30] are similarly not practical when it comes to protecting confidentiality:

- When used in enforcement mode, information flows are blocked, which may break some functionalities. This effectively renders the approach unusable for most end users.

- When used in permissive mode, these tools are unable to track indirect information flows [65].

Figure 5.1 presents our approach to taint tracking for monitoring data leaks. A kernel reference monitor has been implemented in the Linux Kernel and allows for efficient dynamic information flow tracking at the level of system objects (processes, filesystem inodes, etc.).

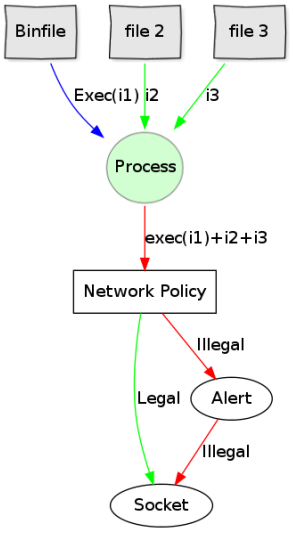

Figure 5.1: Network information flow tracking

Sensitive data is labelled at the filesystem level, and the level of granularity of our approach is at the file level (i.e., files are considered as atomic pieces of information). Our implementation, presented in Chapter 4, takes advantage of the Linux Security Modules (LSM) framework available in the Linux kernel, and taint propagation is triggered by access control hooks. Our design goals are to provide a model that is easy to use, does not lock all the system by default by labelling only the sensitive information, and does not miss any information flow (no false negatives).

5.2 Network extension

We have extended our previous model so as to supervise network interactions. Network sockets are information channels, and we track information flowing towards them. There are different families of sockets, including *UNIX domain sockets* and *internet sockets*. The latter are used to communicate with untrusted remote hosts through the internet, and we focus on their usage by userspace applications. Sockets by themselves are not labelled, as we consider those as part of the process memory. Instead, tracking is performed when processes actually send information through those information channels.

5.2.1 Network policy tag

The policy for communicating with internet sockets is defined globally through a unique shared *network policy tag*. The *network policy tag* is a set of sets defining which combinations of information may legally leave the local system through internet sockets, and optionally which applications may communicate, as well as which information each application may communicate (per-application profiles).

A *network policy tag* is defined as follows:

$$P_{net} \in \wp(\wp(\mathcal{I} \cup \mathcal{X}))$$

It is a set of sets that can contain any combination of elements from $\mathcal{I}$ (passive data) and $\mathcal{X}$ (running code).

The following semantics is associated with P_{net}:

- Elements of $\mathcal{I}$ in the sets of P_{net} represent mutually exclusive sets of data which can legally flow out of the system (i.e., only one of the sets is legal at one time).
- Elements of $\mathcal{X}$ in the sets of P_{net} represent *supervised*[1] code which is allowed to communicate through internet sockets.
- Any combination $A \in \wp(\mathcal{I} \cup \mathcal{X})$ in the sets of P_{net} defines a profile for applications, where elements of $\mathcal{I}$ define which data can be sent over the network, and elements of $\mathcal{X}$ define which running code may send that information.

5.2.2 Legality of network information flows

When a process sends information through a socket, a legality verification is performed on its current *information tag* against the global *network policy tag*. The information flow is legal if and only if the content of its *information tag* is contained in one of the subsets of the *network policy tag*.

Definition 15. For any information tag containing a set of data $S \in \wp(\mathcal{I} \cup \mathcal{X})$, the boolean relation $Legal_{net}$ is defined as follows:

$$Legal_{net}(S) \Leftrightarrow \exists p \in P_{net} | S \subseteq p$$

5.3 Practical use cases

Our approach covers the following use cases. In the following, the term *labelling* refers to the action of attaching a unique *information tag* to a file.

5.3.1 All sensitive data must stay local

In this use case, the user of the system wants all of the sensitive data to stay local. Any network transfer of those data is a violation of the policy and our reference monitor, in its extended version, will report a privacy violation alert. This can be accomplished by only labelling sensitive data (files) that should never flow out of the system. By defining an empty *network policy tag*, no data can legally flow out through network sockets, and the user will be notified every time a socket sends such tainted data over the network.

$$P_{net} = \{\{\}\} = \bot$$

[1] The corresponding binary file is labelled with an *information tag*.

5.3.2 Sensitive data may be sent over the network only through trusted applications

In this use case, the system contains both trusted and untrusted applications, as well as some sensitive data which may flow over the network only through trusted applications. This can be accomplished by labelling all the binary applications on the system along with all the sensitive data. The *network policy tag* is set to match the union of all the *information tags* of the binaries and those of sensitive files on the filesystem. In this case, the *network policy tag* is a set with only one set.

$$P_{net} = (S \cup C)$$

Here S is the set of all the sensitive data and C the set of all trusted code.

5.3.3 Per-application profiles

In this use case, the system contains both trusted and untrusted applications, and each trusted application may send a different set of sensitive data over the network. This can be accomplished by labelling all the binary applications on the system along with all the sensitive data. Then, the *network policy tag* is a set of several sets, where each set represents one application profile, such as:

$$P_{net} = \{\bigcup_{i=1}^{N}(s_i \cup c_i) | s_i \subseteq S, c_i \subseteq C, Legal_{net}(c_i \cup s_i)\}$$

where $Legal_{net}(a \cup b)$ states that the application a is allowed to send information b over the network, as presented in Definition 15.

5.4 Dynamic policy changes

Taint marking can sometimes lead to a growing number of false positives due to the fact that tainted data remains tainted until the system reboots, and information flows keep propagating tainted data between objects of the operating system. This may lead to repetitive alerts about the same data leaking. Furthermore, the user or administrator may decide to declassify some information that he or she previously considered as private, and allow it to flow over the network.

For this reason, users can decide to modify the policy on the fly while the system is running. New sets can be dynamically added to the *network policy tag* at runtime. Several situations may occur:

- Only sensitive data has been labelled, and may not flow over the network. There are no trusted applications. In this case, the user can permanently neutralize alerts concerning a set of sensitive data S by adding a new set S to the *network policy tag*.
- Both sensitive data and trusted application's code have been labelled, and the user wants to neutralize alerts concerning one set of sensitive data S leaked by processes running code C. This can be performed by adding a new set to the *network policy tag* containing $(C \cup S)$.

5.5 Conclusion

In this chapter, we presented a first aspect of our network extension, focussed on tracking outgoing information. We defined a *network policy*, stating how information may leave the operating system through network sockets. The *network policy* can be set on its own, or on top of an information flow policy confining users, applications and persistent containers, as presented in Chapter 3. This extension led to a framework for detecting confidentiality violations through applications leaking information towards the network, which we implemented and evaluated, see Chapter 8.

Chapter 6

Distributed Policy Over Multiple Hosts

This chapter presents our distributed model of intrusion detection. It relies on the host based model that we presented in the first part of this thesis, along with new aspects to take into consideration with respect to the distribution of taint over the network, towards multiple hosts of a supervised network. In the previous part of this thesis, we have shown how using taint marking techniques along with an information flow policy allows us to detect intrusions at the host kernel level. In the previous Chapter, we have extended our host-level so as to track outgoing traffic, and implemented a framework for confidentiality violation detection. In this chapter, we introduce the distributed mechanisms and additions to our model that allow us for intrusion detection in *groups* of supervised hosts.

6.1 Context

In the reminder of this thesis, we propose a distributed model allowing for rich policy specification and fine-grained information flow tracking. We have extended our model in order to detect intrusions in distributed systems composed of multiple hosts gathered in *groups*. Hosts of the same *group* share a common information flow policy. It is distributed in each host at the container level.

In this chapter, we first present the distribution of taint information across all the supervised hosts of a distributed system. After this, we define a distributed information flow policy, allowing to specify the legal behavior of information flows amongst processes of multiple hosts with respect to the involved pieces of information and users on behalf of which processes are running.

Recall from Part II that objects of the operating system such as files, sockets, memory mappings *etc.* are considered as *containers of information* in our model, and that we attach labels to such containers: *information tags*, *policy tags* and *execute policy tags*. Labels are composed of meta-information represented by two sets $\mathcal{I}$ and $\mathcal{X}$, representing respectively passive data and active code. In the first part of our work, labels were containing meta-information specific to the particular host running the IDS.

6.2 Host groups

Distributed systems are generally composed of multiple services running as processes across multiple hosts, involving variable amounts of information. Such information may involve public data, confidential data, executable code *etc.*, from multiple hosts. In order to define an information flow policy for a distributed service, or for a whole distributed system involving multiple services, we gather hosts in *groups* and define a distributed information flow policy per *group*. Let us consider a given network $\mathcal{N}$. Let $\mathcal{H}$ be the set of all hosts on network $\mathcal{N}$. Each host of a *group* is identified by a unique id $h_k \in \mathcal{H}$.

The first step towards defining a distributed policy amongst the hosts of a group is to identify the information to track on each host. Recall from Part II that *information tags* are sets of elements in $\mathcal{I} \cup \mathcal{X}$, identifying passive data and active code residing in containers. For any given host h_k, we define $\mathcal{I}_k$ as the set of all passive data managed by this host, and $\mathcal{X}_k$ the image of $\mathcal{I}_k$ through the *Run* function, *i.e.* the code originating from this host which may be executed by processes on any host. $\mathcal{I}_k$ and $\mathcal{X}_k$ are partitions of respectively $\mathcal{I}$ and $\mathcal{X}$ representing all the information of the *group*:

$$\mathcal{I} = \bigcup_{h_k \in \mathcal{H}} \mathcal{I}_k \wedge \mathcal{X} = \bigcup_{h_k \in \mathcal{H}} \mathcal{X}_k$$

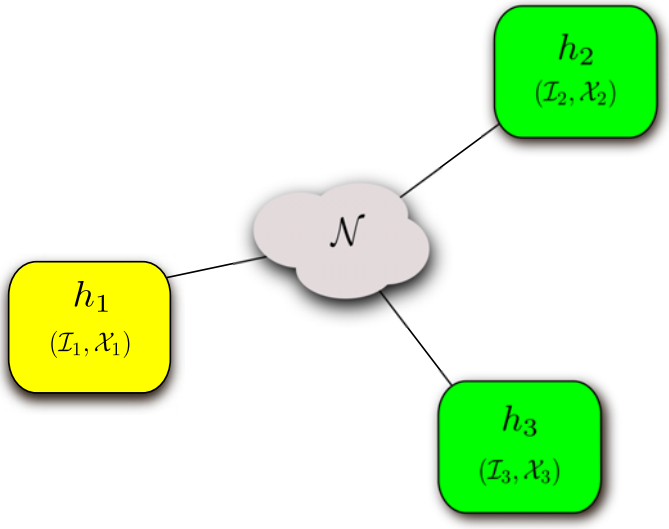

Figure 6.1: Host group

We also define a mapping allowing any host of the group to determine the origin of information: for any piece of tainted information we must be able to determine which host *manages* it, *i.e.* from which host does a specific tainted information come from.

Definition 16. The originating host of an element of $(\mathcal{I} \cup \mathcal{X})$, *i.e.* the host managing a given piece information, is determined by the following relation:

$$Host : (\mathcal{I} \cup \mathcal{X}) \rightarrow \mathcal{H}$$

6.3 Network tainting

Operation	$i \in \mathcal{I}$	$x \in X$
`Read`	taint	discard
`Write`	taint	taint
`Execute`	taint with $x = Run(i)$	discard
`Send`	taint	taint
`Receive`	taint	check legality & discard

Figure 6.2: Tainting rules

Processes are responsible for all information flows[1]. When processes perform actions on objects (*i.e.* other containers), subsequent information flows occur (depending on the operation). The way taint data is carried in our distributed model follows the tainting rules presented in Part II, which apply to all the system objects, with the addition of two new rules, `send` and `receive`, targeting network traffic (through sockets) from and towards other hosts of a *group*, as presented in Figure 6.2. In this figure, *Taint* means that the destination process or container gets tainted by the meta-information. *Discard* means the destination process or container does not get tainted by the meta-information. (For details about the legality of information flows, see Section 6.4).

Hosts from the same *group* exchange information through network messages, which we consider as containers, as well as any other object of the operating system containing information. We therefore attach labels to messages, in order to carry *information tags* between multiple hosts, the same way as we do between containers of the same host. In order to carry taint information, we have considered two methods:

- Embedding information tags as security labels within network messages. This solution can be effective when a small amount of taint data is used. This aspect is further detailed in Chapter 7.
- Translating information tags into security tokens, which can then be resolved between hosts in a peer to peer fashion, using a distributed protocol. We present this method in the next subsection.
- Embedding so-called *deltas* relative to security tokens previously resolved, this is presented later in this chapter.

6.3.1 Distributed security tokens

Information tags can be composed of any amount of meta-information, and thus have dynamic sizes and require variable amounts of space. It may not always be possible to directly represent *information tags* within the labels of networking messages. Therefore, we have introduced a distributed security token protocol allowing hosts of a *group* to exchange security labels in a peer to peer fashion, as shown in Figure 6.3.

Definition 17. Security tokens are images of *information tags* through a cryptographic hash function H as follows, where Θ is the set of all possible tokens:

$$H : \wp(\mathcal{I} \cup \mathcal{X}) \rightarrow \Theta$$

We use such tokens, images of information tags, as security labels on network messages. Recall that *information tags* are dynamic: their content is updated after every information flow. Therefore, processes often need to update the labels they attach to network messages. Our distributed protocol involves so-called *resolvers*, one per host. Resolvers maintain caches of $\langle key : value \rangle$ pairs, storing mappings between *information tags* and *security tokens*. From the point of view of userspace processes, whenever a new token is needed, or an unknown token is received in a network message, a request is made to the local *resolver*.

After every information flow, if the *information tag* of the process has changed, a request is made to the local resolver to create a new token for this process, corresponding to its new *information tag*. We call this step *token creation* in the protocol defined below. Whenever a process receives an unknown token (*i.e.* a token which hasn't been seen before), it needs to query the local resolver, which in turn will query the originating host in order to receive a mapping, in the form of a $\langle key : value \rangle$ pair, associating the requested token with an *information tag*. We call this step *token resolution*. Such a mapping allows to translate the new token into an information tag, and to taint the process which received the network message accordingly. Local *resolvers* run on each machine, represented as $\mathcal{R}_1$ and $\mathcal{R}_2$ in Figure 6.3, and are the only processes communicating with no security labels, *i.e.* the code of resolvers is trusted and we do not track information flows

[1] processes are the only *active* objects of the system: the execution of any pieces of code is necessarily performed through a process.

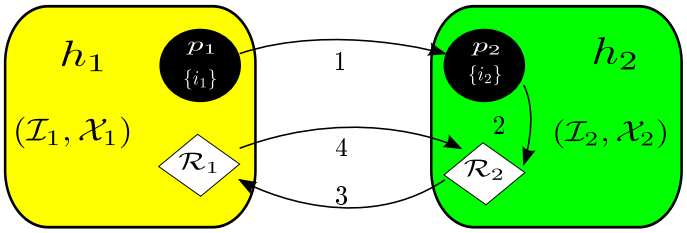

Figure 6.3: P2P token exchange

between the resolvers of multiple hosts. In practice, the code of the resolvers needs to be audited and verified against security flaws.

Figure 6.3 summarizes the steps involved in the protocol:

1. A message containing a security token is sent by a userpace process p_1 on host h_1 to another process p_2 on host h_2.
2. Process p_2 asks the local resolver $\mathcal{R}_2$ to look up in its local cache in order to resolve this token.
3. If it cannot find it, $\mathcal{R}_2$ asks $\mathcal{R}_1$ for resolution using the protocol defined in the next section.
4. $\mathcal{R}_1$ replies to $\mathcal{R}_2$ with a mapping. $\mathcal{R}_2$ is now able to resolve the new token for p_2.

6.3.2 Protocol

Resolvers maintain a local cache of sent and received tokens for each remote host of the *group*, as shown in Figure 6.4. For any pair of hosts (h_1, h_2), we name the caches as follows:

- $sent_{h_1}[h_2]$ is the cache of tokens sent to host h_2, on host h_1.
- $recv_{h_1}[h_2]$ is the cache of tokens received from host h_2, on host h_1.

The sizes of sent and received caches are equal and noted ℓ. Caches contain $\langle key : value \rangle$ pairs in $(\Theta \times \wp(\mathcal{I} \cup \mathcal{X}))$. Token resolution and token creation (described below) ensure that for each pair of hosts (h_1, h_2), $sent_{h_2}[h_1]$ is synchronized with $recv_{h_1}[h_2]$. Token resolution is performed over an alternate secure channel, using unlabeled messages (*i.e.* no security labels). Possible operations on both caches are:

- Creating a new entry (C).
- Overwriting an existing entry (O).
- Reading an existing entry (R).

a. Token resolution: when a process receives a network message labeled with a security token, it needs to *resolve* it in order to be able to append the appropriate taint data to its *information tag*. Token resolution is defined by the following relation:

$$resolve : \Theta \rightarrow \wp(\mathcal{I} \cup \mathcal{X})$$

Token resolution can be done directly by the local resolver if the token is in the local cache of received tokens. Otherwise, it is necessary to query the remote host using the protocol defined below.

b. Token creation: when a process on host h_{local} updates its *information tag*, the following actions are necessary before sending data to any destination host h_{dest}.

1. Create a new token $tk_{new} = H(itag)$ where $itag$ is the current *information tag* of the process, and H is a cryptographic hash function (see Definition 17).

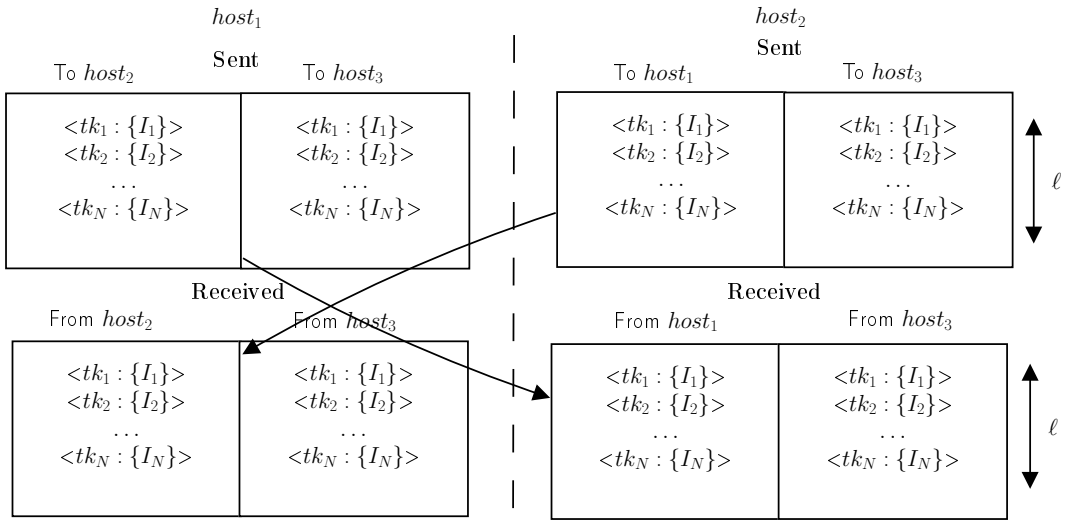

Figure 6.4: Distributed token protocol

2. Read local cache entries (R) and check for collisions[2] with tk_{new}: $sent_{h_{local}}[h_{dest}][tk_{new}]$ may already exist as a key for a different *information tag*.

3. In case of collision, overwrite (O) existing values and set $flag$ to `O_REPLACE`.

4. Otherwise, create (C) tk_{new} in $sent_{h_{local}}[h_{dest}]$ and set flag to `O_NEW`.

5. Send token to remote host (using the protocol defined below and the appropriate flag).

c. Token exchange protocol: hosts of a *group* exchange tokens using a protocol based on the two following operations:

- Function $token\_query(token, host)$: query *host* about *token*. The remote host replies with $token\_send$ and sets a flag to either `O_NEW` or `O_REPLACE`. When `O_REPLACE` is set, a previous cache entry with the same key already exists and must be replaced. Otherwise, it is a new entry.

- Function $token\_send(token, flag, host)$: send the pair $(token, H(token))$ to *host* with $flag$ in {`O_NEW`, `O_REPLACE`}.

6.3.3 Frequent updates

The overhead of the protocol that we defined depends on how often information tags of communicating processes require updates. Considering that our model does not yet have support for declassification[3] the behavior of information tags is such that, for a given process, it can only grow in size, and never diminish, until the process gets killed (or respawned). New elements may be added to the information tags, but no elements may be removed.

Definition 18. Whenever a process p receives a network message m, while in state i, we compute the update information tag of p as follows:

$$itag(p)_{i+1} = itag(p)_i \cup itag(m)_i$$

[2]Even though collision probability is extremely low, it may occur, as in practice, H is a hash function.

[3]In our model, the support for declassification would refer to the ability for users, programs or containers to declassify information based on rules defined in the information flow policy. This could be done, for instance, by untainting some information given such rules, or by tainting it with new identifiers.

As opposed with labeling a connection between two hosts, we label each network message individually, based on the state of processes at the moment when each message is sent. One possible drawback of this approach is when small updates are performed frequently on a process's information tag (*e.g.* because it accesses new tainted files before sending each message). In this case, the performance overhead of the resolution protocol would increase considerably. In order to avoid such a problem, we compute *deltas*.

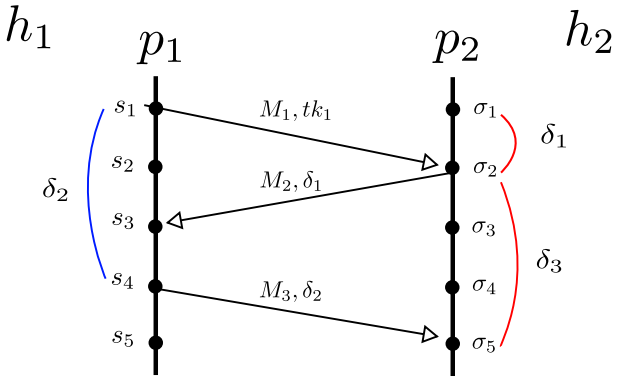

Figure 6.5: Computing deltas

Deltas contain the difference between two information tags (*i.e.* their union minus their intersection), or the difference between two states of the same information tag, *e.g.* the information tag of the process p_1 in state i and in state $i_4 = i + 4$. We embed *deltas* directly within network messages, in the security labels, when possible. It is not always possible due to size restrictions, therefore, *deltas* are used when "small" changes happen in information tags, *i.e.,* when the memory space required to represent the new elements to append does not exceed the maximum size of security labels.

Definition 19. We define the Δ relation as follows, returning the delta between two information tags (sets of elements of $\wp(\mathcal{I} \cup \mathcal{X})$):

$$\Delta(a, b) = \{x | x \in a \wedge x \notin b\}$$

Definition 20. Let us define the maximum available space in a network message security label as λ_{lbl}, and the size (space) of individual elements of information tags as λ_{tag}. Labelling network messages with lambdas rather than with security tokens is preferred whenever the available space in network messages is sufficient, *i.e.* whenever:

$$|\Delta(itag(p)_t, itag(p)_{t+k})| \times \lambda_{tag} < \lambda_{lbl}$$

Example 15. Figure 6.5 shows an example of two communicating processes p_1 and p_2. For each process, bullet points represent the different states of their information tags. Process p_1 has states s_1 to s_5 and p_2 has states σ_1 to σ_5. We consider the initial states s_1 and σ_1 synchronized with respect to the token caches: $sent_{h_1}[h_2]$ is synchronized with $recv_{h_2}[h_1]$ and $sent_{h_2}[h_1]$ is synchronized with $recv_{h1}[h_2]$.

1. Process p_1 sends a message M_1 to p_2, labelled with the token tk_1.
2. Due to network latency, p_2 is in a new state σ_2 when it receives the message. However it can directly resolve the token tk_1, as this one is present in the local cache of received tokens.
3. Process p_2 now sends a message M_2 to p_1. As the state of p_2 has changed from σ_1 to σ_2, it is required to compute $\delta_1 = \Delta(itag(p_2)_{\sigma_1}, itag(p_2)_{\sigma_2})$. As this is a minor change, between two consecutive states of process p_2, δ_1 can fit in M_2.
4. Process p_1 receives M_2. Its information tag has changed two times since it sent M_1, and it is now in state s_3. The delta since state s_1 is d_1. It uses δ_1 to update its information tag with the information in M_2, and jumps to state s_4.

5. Process p_1 sends message M_3 to p_2. In the local cache $sent_{h_1}[h2]$, the last sent token is tk_1, p_1 changed of state 4 times since then. It now computes $\delta_2 = \Delta(itag(p_1)_{s_1}, itag(p_1)_{s_4})$ and check whether it fits in the message label (see Definition 20). As it fits, M_3 is labeled with δ_2.

6. Upon reception of M_3, p_2 reads delta δ_2 and update its *information tag*.

6.4 Information flow policy

In Chapter 3, we have introduced the information flow policy of our model at the host level. Processes run code (or programs) on behalf of users. Recall from Chapter 3 that their policy tags are determined dynamically by $\mathbb{P}_U$ and $\mathbb{P}_\Pi$, at execution time, as the intersection of the policy attached to the user on behalf of which the program is being executed, and the policy attached to the executed program (*i.e.* $\mathbb{P}_u \sqcap \mathbb{P}_\pi, u \in \mathcal{U}, \pi \in \Pi$). When dealing with multiple hosts gathered in *groups*, we need to take new aspects into consideration. With a distributed information flow policy, each local information flow may involve tainted information from any host of the *group*. The local information flow policy on each host therefore refers to such disperse information, as well as how it may flow from one host to another. It involves users, active code (programs), persistent containers (*e.g.* files) and network sockets.

Definition 21. The local information flow policy on any host h_i of a *group*, is expressed independently for users, active code (programs), persistent containers (*e.g.* files), and network packets, and is specified in the *policy tags* of containers, in a decentralized manner. It is defined by the quadruplet $\mathbb{P}_{h_i} = (\mathbb{P}_{\mathcal{C}_P}, \mathbb{P}_\mathcal{U}, \mathbb{P}_\Pi, \mathbb{P}_{Net})$ where:

- $\mathbb{P}_{\mathcal{C}_P}$ is the set of all the policy tags restricting passive containers (mostly files).
- $\mathbb{P}_\mathcal{U}$ is the policy restricting local users.
- $\mathbb{P}_\Pi$ is the policy restricting executable code.
- $\mathbb{P}_{Net}$ is the policy restricting network communication (as presented in Chapter 5).

Definition 22. The information flow policy $\mathbb{P}_{group}$ of a group of hosts $(h_1, \cdots, h_N)$, identified at each host's level is defined as

$$\mathbb{P}_{group} = (\mathbb{P}_{h_1}, \cdots, \mathbb{P}_{h_N})$$

The following properties can be expressed (any number of each), and verified by the reference monitor of each host.

6.4.1 Users

Users in our model refer to the users on behalf of processes (in the UNIX sense). The following properties may be expressed in the information flow policy so as to restrict the behavior of processes towards information (data or code) from other hosts of the same *group* with respect to local users on each machine.

$$\forall u \in \mathcal{U}, \mathbb{P}_u \subseteq \wp(\bigcup_{k=1}^{N}(\mathcal{I}_k)) \tag{6.1}$$

(6.1) specifies the following properties:

- Local user u may only *access* the specified pieces of information from hosts $h_1, \ldots, h_k$ within the group[4], (*secrecy w.r.t.* users).
- Local user u may only *mix together* the specified piece of information from host $h_1, \ldots, h_k$ within the group (*integrity w.r.t.* users).

[4]Including the local host, this goes for all the other properties as well.

$$\forall u \in \mathcal{U}, \mathbb{P}_u \subseteq \wp(\bigcup_1^k(\mathcal{X}_k)) \tag{6.2}$$

(6.2) specifies the following property:

- Only the specified pieces of code from hosts $h_1, \ldots, h_k$ may be *executed* by user u on the local host (*execution w.r.t. users*).

Properties 6.1 and 6.2 may be used together in the policy. In this case, the resulting policy contains sets of elements of $\mathcal{I}$ and $\mathcal{X}$, *i.e.* $\mathbb{P}_u \subseteq \wp(\bigcup_1^k(\mathcal{I}_k \cup \mathcal{X}_k))$.

6.4.2 Programs

Programs refer to the active code being run by processes. Once (optional) rules have been defined for user accounts (users, on behalf of which processes are being executed), the information flow policy may also contain the following properties, specifying rules attached to pieces of active code being run by processes (individual or multiple elements of $\mathcal{X}$ forming any program $\pi \in \Pi$).

$$\forall \pi \in \Pi, \mathbb{P}_\pi \subseteq \wp(\bigcup_1^k(\mathcal{X}_k)) \tag{6.3}$$

(6.3) specifies the following property:

- Local processes running π as code on the local host may only *execute* the specified sets of information from hosts $h_1, \ldots, h_k$ of the group (*execution w.r.t. running code*).

$$\forall \pi \in \Pi, \mathbb{P}_\pi \subseteq \wp(\bigcup_1^k \mathcal{I}_k) \tag{6.4}$$

(6.4) specifies the following properties:

- Local processes running π as code may only *access* the specified pieces of information from hosts $h_1, \ldots, h_k$ within the group (*secrecy w.r.t.* programs).
- Local processes running π as code may only *mix* the specified pieces of information from hosts $h_1, \ldots, h_k$ together (*integrity w.r.t.* programs).

Properties 6.3 and 6.4 may be used together in the policy, leading to $\mathbb{P}_\pi \subseteq \wp(\bigcup_1^k \mathcal{I}_k \cup \mathcal{X}_k)$.

6.4.3 Persistent containers

Persistent containers are individually protected by the following properties.

$$\forall c \in \mathcal{P}_\mathcal{C}, \mathbb{P}_c \subseteq \wp(\bigcup_1^k \mathcal{I}_k) \tag{6.5}$$

(6.5) specifies the following properties:

- Persistent container c may only *contain* the specified pieces of information from hosts $h_1, \ldots, h_k$ within the group (*secrecy w.r.t.* persistent containers).
- Persistent container c may only *mix* the specified pieces of information from hosts $h_1, \ldots, h_k$ together (*integrity w.r.t.* persistent containers).

$$\forall c \in \mathcal{P}_\mathcal{C}, \mathbb{P}_c \subseteq \wp(\bigcup_1^k \mathcal{X}_k) \tag{6.6}$$

(6.6) specifies the following properties:

- Write (or append) access to the persistent container c is only authorized to processes running the specified code, from hosts $h_1, \ldots, h_k$ within the group (*integrity* of containers *w.r.t.* running code).

Properties 6.5 and 6.6 may be defined together, leading to $\mathbb{P}_c \subseteq \wp(\bigcup_1^k \mathcal{I}_k \cup \mathcal{X}_k)$.

6.4.4 Network packets

Incoming and outgoing traffic is tracked at the network packet level. The following properties may be expressed in $\mathbb{P}_{Net}$ so as to restrict incoming or outgoing traffic.

$$\mathbb{P}_{Net} \subseteq (\wp(\bigcup_1^k \mathcal{I}_k) \tag{6.7}$$

- In the case of incoming traffic, only the specified sets of information from hosts $h_1, \ldots, h_k$ are allowed in.

- In the case of outgoing traffic, only the specified sets of information from hosts $h_1, \ldots, h_k$ are allowed out.

$$\mathbb{P}_{Net} \subseteq \wp(\bigcup_1^k \mathcal{X}_k) \tag{6.8}$$

- In the case of incoming traffic, only accept traffic from remote processes running the specified sets of code (programs).

- In the case of outgoing traffic, only accept outgoing traffic from local processes running the specified sets of code (programs).

Both properties 6.7 and 6.8 may be used together, resulting in:

$$\mathbb{P}_{Net} \subseteq \wp(\bigcup_1^k \mathcal{I}_k \cup \mathcal{X}_k) \tag{6.9}$$

6.5 Legality of information flows

Recall definition 18 and Figure 6.2 from this Chapter, defining tainting rules with respect to the different objects of the operating system. Such rules apply after each *operation* responsible for information flows, made by processes running code on behalf of users. The legality of such information flows depends on the updated *information tags* with respect to the local information flow policy on the local host, on each host of the *group*. Therefore, information flows between several hosts involve the local information flow policies of each host (subsets of $\mathbb{P}_{group}$). An information flow towards any container c is legal if and only if its new *information tag* $itag(c)$, after the information flow occurred, is included in at least one of the sets of its *policy tag* $ptag(c)$. This applies to all kinds of containers (*i.e.* processes as well as passive containers and sockets), based on the properties defined above. In a group involving k hosts, $itag(c) \in \wp(\bigcup_1^k \mathcal{I}_k \cup \mathcal{X}_k)$ and $ptag(c) \subseteq \wp(\bigcup_1^k \mathcal{I}_k \cup \mathcal{X}_k)$. This is verified by the relation *Legal*, defined in Part II (Definition 3), which can be generalized as follows:

$$Legal(A, B) \Leftrightarrow \exists a \in A | a \subseteq B$$

6.5.1 Policy tags

In our model, *policy tags* are the link between the information flow policy with its different aspects, or subsets, and the objects of the operating system we actually supervise at runtime. Each policy tag contains rules, that are part of either $\mathbb{P}_{\mathcal{P}_C}, \mathbb{P}_{\mathcal{U}}, \mathbb{P}_{\Pi}$ or $\mathbb{P}_{Net}$. In Chapter 3, we introduced the notion of *policy tags* of persistent containers and processes, and how these two relate to different subsets of the information flow policy. Recall from previous chapters that the policy tags of processes are dynamically set up at runtime, upon process creation, from $\mathbb{P}_{\mathcal{U}}$ and $\mathbb{P}_{\Pi}$. The policy of persistent containers is initially attached to their respective policy tags, and expressed from rules of $\mathbb{P}_{\mathcal{P}_C}$. Similarly, the network policy tag, directly equal to $\mathbb{P}_{Net}$, is attached to network sockets so as to track incoming and outgoing network packets. However, it is common to all sockets, regardless of which process created them. The reason for this is that processes each have their own policy tag already. $\mathbb{P}_{Net}$ is intended to track incoming and outgoing traffic based on the properties defined above.

	$\mathbb{P}_{\mathcal{P}_C}$	$\mathbb{P}_{\mathcal{U}}$	$\mathbb{P}_{\Pi}$	$\mathbb{P}_{Net}$
Processes		✓	✓	
Persistent containers	✓			
Sockets				✓

Figure 6.6: Deriving policy tags from the policy

6.6 Conclusion

In this chapter, we have shown how we extended our information flow model to distributed systems made of multiple hosts gathered in *groups*. Security labels are carried over the network, and we are now able to define the legal interactions between processes of different hosts, given their underlying $\{user, code\}$ context. The information flow policy is distributed in a peer to peer fashion, and hosts exchange security labels through a distributed token protocol. As our model may involve frequent updates of security labels in some situations, we propose a solution to diminish the stress on the token protocol by computing *deltas*, containing the relative difference between the states of the information tags of communicating processes. The information flow policy is defined at each host level, in the information tags of processes, persistent containers and sockets, as shown in Figure 6.6. It is verified by each host kernel, the only trusted code in our model. This extended model and its implementation represent our second contribution. Related work include Aeolus [13], DStar [77] and Pedigree [74]. Our approach differs from these in multiple manners:

- Aeolus is a framework for building secure applications. It tracks information flows at the thread level and allows users to restrict the use of their information, which is categorized in tags. Such a framework offers finer-grained information flow tracking than our approach (we work at the system object level, *e.g.,* processes instead). However, this framework does not provide system-wide supervision, and it is not applicable to native applications. Its policy definition is user-centric and offers limited expressiveness.

- DStar [77] is an extension of decentralized information flow control models such as Flume [40] and Histar [76], to distributed systems. It uses *integrity* and *secrecy* labels so as to categorize information and restrict processes. It ensures that only processes allowed to communicate may actually do so, and follows a "no read-up, no write-down" logic. As other decentralized information flow control models, DStar requires applications to be modified in order to benefit from the declassification mechanisms that it offers. DStar uses exporter daemons along with cryptographic certificates to exchange security labels between hosts. Our approach uses similar mechanisms (though it lacks mechanisms to enforce the integrity of security labels in its current state) to exchange security labels amongst hosts. However, both our labels

and our security policy differ from DStar. On the one hand, our labels contain unlimited taint information elements, each describing one individual piece of information. On the other hand, the definition of the policy in our model does not rely on security classes but instead attaches individual sets of rules to each piece of information, in a fine-grained manner. This allows us to track illegal execution of code as well as *integrity* or *confidentiality* violations by users or applications.

- In Pedigree, taint information is attached to resources such as files. Taint information may of two kinds: secrecy and integrity. As DStar, the policy is based on a lattice. However, as opposed to DStar, the policy is centralized. The particularity of Pedigree is that it provides *capabilities* mechanisms attached to taint information, so as to provide declassification. Our approach does not offer declassification mechanisms, but it keeps information flow histories and allows to define fine-grained policies (as described in the previous paragraph).

To the best of our knowledge, the model that we have introduced is the only anomaly detection model combining OS-level taint analysis along with a fine-grained policy definition so as to detect intrusions in distributed systems. In the next chapters, we present the implementation of this model as well as our experiments along with a discussion of the strong and weak points of our approach (in Section 8.4).

Chapter 7

Network and Distributed Implementation

This chapter presents our distributed implementation. It presents the additions we added in the previous implementation in order to take the network and distributed aspects of the model into account. This implementation is an extension of the implementation presented in Chapter 4. It adds support for security labels on network sockets, along with a network policy, checking that only processes allowed to do so may leak the specified information, as presented in Chapter 5.2. Furthermore, it takes new distributed aspects into consideration, by labelling individual network packets with security information (information tags), so as to carry taint information between hosts of a *group*.

7.1 Network policy

The *Network policy tag*, that we introduced in Chapter 5.2, is used to track outgoing traffic through internet sockets, by specifying which processes (more exactly, which pieces of executed code) are allowed to communicate information out of the system. A single network policy tag is centrally defined for all the system. As the other policy tags, attached to other types of containers, it is specified as a set of sets:

$$\mathbb{P}_{net} \in \wp(\wp(\mathcal{I} \cup \mathcal{X}))$$

It is implemented as a linked list of legal sets of information, where each set is stored in a red black tree for fast $o(log(n))$ lookups. A userspace interface is exported through the securityfs[1] filesystem, in order to load the *network policy tag* in kernel memory at boot time. This interface is accessible through `sys/kernel/security/blare/network` (once the securityfs filesystem has been mounted, *e.g.* by adding the correct line to `/etc/fstab`). Userspace tools have been written to set and update the network policy at runtime, and are available for download at `http://blare-ids.org`. These tools add new sets of information, one at a time, to the network policy. Each set is represented in userspace as an array of integers, contiguous in memory. The kernel code receipts the data and converts it into `blare_policy_tree` elements.

[1] Securityfs is a pseudo filesystem based on sysfs and is used by the LSM modules, generally mounted as /sys/kernel/security.

```
/* A policy tag is a list of binary trees (a set of sets)
 * Each binary tree has the following type:
  * */
  struct policy_tree{
        struct list_head list;
        struct rb_root *root;
        int cardinal;
  };
```

Such elements make use of the SLAB for efficient memory allocation (see Chapter 4). The relevant code is defined in `security/blare/network.h`.

The network policy applies to both socket families `AF_INET` and `AF_INET6`, respectively related to IPv4 and IPv6. A userspace daemon reports alerts to the user via the libnotify library (by checking the output of the system logs for entries written by our reference monitor).

7.2 Distributed policy

The distributed version of this implementation carries security labels on network packets, so as to transfer taint information between hosts of the same *group*. The information flow policy for the group is distributed in each host, at the container level. Hosts are able to determine the origin (*i.e.* which host it came from) of each piece of information, and the local policy tags of container determine their legality. The network policy, as defined in the previous section, can be used to track incoming and outgoing information on each host of the group. Therefore, the legality of information flows towards processes is verified in a two way run:

- First, on packet reception, the reference monitor checks the content of the security label, and verifies that it is legal with respect to $\mathbb{P}_{Net}$.
- Then, the data content of the label (elements of $\mathcal{I}$ only[2]) is merged with the information tag of the process, and the updated information tag's legality is verified. This process is the same as when reading files: elements of $\mathcal{X}$ are discarded, see Chapter 3.

7.3 CIPSO

To achieve the transportation of security labels over the network, we use CIPSO labels: CIPSO[3] is an IETF draft proposing a "Commercial IP Security Option". It defines a type of security options for IPv4 packets. Note that we do not support IPv6 yet in this implementation. Existing efforts to support labels on IPv6 packets include the Common Architecture Label IPv6 Security Option (CALIPSO), however no support for any such option exists in the Linux kernel at this time. As

Type 134	Option Length	Domain of interpretation	Tags
8 bit	8 bit	32 bit	272 bit

← 320 bit →

Figure 7.1: CIPSO option

shown on Figure 7.1, CIPSO option size is limited to 40 bytes (320 bits), the current limit for IPv4 options. The tags field is used to pass the actual security information related to the packets

[2] Recall that elements of $\mathcal{X}$ in a process's information tag refer to the code currently being run. Similarly, elements of $\mathcal{X}$ on a network packet security label refer to the code being run by the process which sent the message. Therefore, merging such elements in the information tag of processes on packet reception would make the new tag inconsistent.

[3] `https://tools.ietf.org/html/draft-ietf-cipso-ipsecurity-01`

content, describing so-called *categories* (in our case, categories refer to the meta-information of *information tags*). The Domain of interpretation (DOI) field gives the ability to define separate domains where categories may have different meanings, *e.g.* for some systems, a value of 5 in the tags may be equal to the MLS level "top secret", where in some other domains, it could be interpreted as "public".

Tag type	Tag length	Tag Information
8 bit	8 bit	256 bit

Figure 7.2: Tag types

CIPSO allows for up to 128 tag types, however the current draft only defines types 1, 3 and 5.

- Tag type 1 defines a bitmap of *categories*, (*i.e.* values representing information in the packets) from category 0 to category 239.
- Tag type 3 defines a set of enumerated categories (*i.e.* representing sparse values).
- Tag type 5 defines categories ranges, where each range includes multiple categories.

Tag type	Tag length	Alignment Octet	Sensitivity Level	Bitmap of Categories
8 bit	8 bit	8 bit	8 bit	240 bit

Figure 7.3: Tag type 1

In our implementation, *information tags* are sets of 32 bit integers, thus we cannot have more than 10 *information tag* elements per IP option if we directly taint network packets with *information tags*. In order to overcome this limitation, we have designed a distributed security token management protocol, allowing any host of a *group* to securely exchange security labels, as presented in Chapter 6. However, for the sake of simplicity, our current implementation labels network packets directly by using the tag type 1 as defined in CIPSO, as shown on Figure 7.3. By using a bitmap, we are able to represent up to 240 distinct information tags, and thus track up to 240 distinct information elements (including data and code) per *group* of supervised hosts, which lets us track a sufficient amount of taint information for realistic experiments. Therefore, each host $h_1, \ldots, h_N$ of the group has reserved space in the bitmap to represent its local information $\mathcal{I}_N$ and code $\mathcal{X}_N$.

7.4 Netlabel

CIPSO labels are supported in the Linux Kernel, through the NetLabel subsystem. NetLabel provides an API for LSM modules to attach CIPSO labels to outgoing or incoming network traffic generated by userspace applications. The API provides functionalities exported to LSM modules, translating operations on packets into low level protocol operations. This is defined in the kernel source, in the header file `include/net/netlabel.h`.

7.4.1 Internal representation

The main structure that is used by NetLabel to represent security information is the following:

```
struct netlbl_lsm_secattr {
        u32 flags;
        [...]
        u32 type;
        char *domain;
        struct netlbl_lsm_cache *cache;
        struct {
                struct {
                        struct netlbl_lsm_secattr_catmap *cat;
                        u32 lvl;
                } mls;
                u32 secid;
        } attr;
};
```

This structure contains the necessary fields to represent a CIPSO option. It embeds a so-called *category mapping* in `struct netlbl_lsm_secattr_catmap *cat;`. This latter structure is used to represent the tags. Labels can be attached and removed from sockets. When a label is attached to a socket, all the packets leaving the system through this socket are labelled with it. The following functions are used to set or remove a label on a socket:

```
static inline int netlbl_sock_setattr(struct sock *sk, u16 family, const struct
                    netlbl_lsm_secattr *secattr);
static inline void netlbl_sock_delattr(struct sock *sk);
```

It is also possible to directly label network packets, by using the following function:

```
static inline int netlbl_skbuff_setattr(struct sk_buff *skb,
                    u16 family,
                    const struct netlbl_lsm_secattr *secattr);
```

7.4.2 Conversion

In order to convert from and to NetLabel CIPSO bitmap representation into *information tags* (*i.e.* doubly linked lists of integers, see Chapter 4) as used in our model, we defined two functions in `security/blare/netlabel.c`:

```
struct list_head *blare_catmap2itag(struct netlbl_lsm_secattr_catmap *catmap);
int blare_itag2catmap(struct list_head* itag, struct netlbl_lsm_secattr_catmap
                    *catmap);
```

These two functions respectively convert a category mapping in the form of a 240 bit bitmap into a linked list of 32 bit integers, and the other way round. It allows us to embed bitmaps into outgoing packet headers using CIPSO option type 1, and to retrieve them from incoming packets. The LSM hooks that we use for this purpose are presented later in this chapter.

7.5 Execution contexts

Before going into further details about how we have implemented network information flow tracking, let us introduce the notion of *execution contexts*. Kernel code may run in different contexts. When executing code on behalf of a userspace process (*e.g.*, executing a system call), it runs in *process context*, where it has access to all the data structures of the current userspace process. The code running in process context can sleep (and be rescheduled later). Most aspects of this implementation run in *process context*. Networking code, however, is often related to low level data

structures, involving time critical operations, *e.g.*, copying data from the network card buffers into memory on reception of packets. When a piece of hardware uses an interrupt to notify the CPU about some event, the CPU immediately schedules the appropriate interrupt handler (based on the interrupt number). When executing an interrupt handler, the kernel is in *interrupt context*. This context is not attached to any process (though the address space of the interrupted process is left as-is), and the code cannot sleep. Interrupt handlers may interrupt important code, including other interrupt handlers, or may disable all other interrupts for the time of their execution. For these reasons, interrupt context code has to run for the shortest possible time. Therefore, the processing of interrupts is split in two parts: *top half* and *bottom half*. The interrupt handler is the *top half*, and it only processes immediate and time critical operations. All the remaining processing is left for the *bottom half*, generally deferred in a softirq or in a tasklet. We will not go into further details about these inner mechanisms, please refer to Robert Love's "Linux Kernel Development" book [44], or "Understanding the Linux Kernel" by Daniel Bovet and Marco Cesati [9] for a more comprehensive description.

7.6 Socket operations

In Chapter 4, we showed how network traffic between local processes is tracked, involving UNIX sockets (`AF_UNIX`) or internet sockets (`AF_INET`). We will now present mechanisms to track information between processes of different hosts of the same group by using CIPSO labels.

7.6.1 Sending messages

We attach labels to outgoing network packets by using the LSM hook `security_socket_sendmsg`, hooking the function `__sock_sendmsg()` in `net/socket.c`. This hook calls back functions in our LSM module, and the code runs in the context of the userspace process which called the `sendmsg` system call. Whenever the destination host is different from the local host, information tags are converted into bitmaps at this stage, and embedded into the network packet using the Netlabel LSM API. Otherwise, the information tag of the socket itself is labelled, as presented in Chapter 4. The relevant code is defined in `security/blare/lsm_hooks.c`. In order to avoid concurrent access to to the underlying `sock` structure, attached to the socket, we need to take extra precaution when attaching a security label to it.

```
local_bh_disable();
bh_lock_sock_nested(sk);
rc = netlbl_sock_setattr(sk, sk->sk_family, &secattr);
bh_unlock_sock(sk);
local_bh_enable();
```

The `local_bh_disable()` macro disables bottom halves on the local CPU. This ensures that we are not interrupted by a *softitq*, like those triggered by the reception of network packets (see next Section). However, bottom halves may still execute on other CPUs, therefore we also need locking on the sock structure, and this is what the macro `bh_lock_sock_nested()` does by disabling the preemption (by calling `preempt_disable()`) and holding a spinlock.

7.6.2 Receiving messages

Incoming traffic is tracked with the LSM hooks `security_sock_rcv_skb` and `security_socket_recvmsg`. The former is called on frames reception, just after those get attached to the related socket. The code calling this first hook does not run in the context of the userspace process which received the message. In other words, we do not have access to the data structures related to the process receiving the message. This is due to the fact that receiving messages is done in *interrupt context*. The interrupt handler copies the packet (or frame) in an `sk_buff` structure and initializes some other data structures before notifying the kernel about the new received frame, and deferring further processing to a softirq. The

hook `security_sock_rcv_skb` is triggered by the function `sk_filter()` in `net/core/filter.c`, filtering socket buffers. The caller of this hook holds spinlocks and runs in a *softirq*, therefore the code from our module that is called back at this very instant cannot sleep (otherwise resulting in a catastrophic behavior, most likely ending up as a kernel panic). Note that there are no mechanisms avoiding the same softirq to run concurrently on several CPUs, therefore specific precautions have to be taken so as to avoid concurrency issues. Furthermore, when allocating kernel memory in such a context, one needs to make sure that the `GFP_ATOMIC` flag is used, so as to avoid the underlying call to `get_free_pages()` to sleep. In this part of the code, we perform the following operations:

1. Dump the security attributes attached to the headers of the packet. This is done by calling `netlbl_skbuff_getattr()` from the netlabel API.
2. Acquire a spinlock on the socket's tags. `blare_tags` structures (see Figure 7.4 below) are attached to sockets in their `sk->sk_security` field. This ensures that no concurrent *softirq* running the same code accesses the same data structure at the same time. Note that we do not disable local bottom halves here, on the first hand because softirqs never preempt each other (only interrupt handlers may preempt softirqs), and on the second hand because the only possible concurrent code in this situation is the same softirq running on another processor, which is solved by the spinlock.
3. Extract the bitmaps from network packets and make the conversion into the *information tag* of the socket.
4. Release the spinlock.

```
/* Set of tags to attach to any object */
struct blare_tags{
    struct list_head *info;
    /* Used by softirqs invoking rcv_skb*/
    spinlock_t info_lock;
    atomic_t refcount;
    int info_rev; //unused
    struct list_head *policy;
    struct list_head *xpolicy;
};
```

Figure 7.4: The `blare_tags` structure, attached to sockets (and other objects)

At this point, the information tag of the socket, stored in the `info` field of the socket's tags, is up to date. We now have to update the information tag of the process which received the message, with the socket's information tag. This is done by a second hook, `security_socket_recvmsg`. This part of the code does:

1. Get a copy (RCU) of the current process's information tag.
2. Merge the socket's information tag into this copy of the process's information tag.
3. Commit (RCU) the new information tag (which replaces the current process's information tag with the new one).

No specific precautions are required here, as this code run in process context: we can access the relevant data structures directly (which we could not in the previous hook), and we can safely sleep (no precautions regarding memory allocation or specific function calls). Furthermore, though the two hooks may run concurrently (*i.e.* a new frame may arrive in the network card buffer, deferring work in a softirq, triggering the first hook, while the code called by the second hook runs on another CPU), this code is perfectly safe without any locking. This is due to the fact that information tags

of sockets are, like those of processes, implemented as doubly linked lists. Such data structures are safe in the case of concurrent readers and writers, as long as there is no more than one writer at the same time.

7.7 Bug and patch

During the development of our kernel monitor, we stumbled across an issue due to a bug in the code of the kernel, outside our module in the Netlabel subsystem. Our testing environment was composed of several virtual hosts running our modified kernel, connected over a virtual bridged network. The host kernel was the default Debian kernel. In our test case, all the packets containing a CIPSO label were dropped by the host kernel. After a period of testing and discussion with the author[4] of the code, we could figure that this was due to a bug in the code of Netlabel and identify possible ways to reproduce it. A patch has been released by Paul Moore to fix this bug[5], and it was accepted in the Linux kernel in version 3.5-rc1. Before this patch, it was required that the host kernel be configured to use netlabel with the same domain of interpretation as the guests. Not doing so was resulting in a host kernel failing to route network packets in the case of bridged networks.

7.8 Future work

In the future, several optimizations and new features should be considered, so as to increase performance and stability to a higher level.

7.8.1 Distributed security token

We have not implemented the distributed security token protocol presented in Chapter 6, this is left for future work. Therefore, the current implementation has a limitation on the number of distinct meta-information that can be carried on network packets. The protocol we defined can be implemented using netlink messaging [52], so as to communicate with a user space daemon on each host. Labels resolution would then be performed by the local userspace daemon towards the remote daemon in a peer to peer fashion every time a new and unseen token arrives in a network packet.

7.8.2 Copy on write

Information tags of processes, sockets, shared memory segments, and every other objects represented in memory, should be implemented using *copy on write* so as to reduce the memory overhead of our reference monitor. Objects of the system tainted by the same information tag should hold a pointer to the same data structure rather than a copy of it, until it needs to modify it to add new taint data. A cache could be used to maintain all existing information tags in the system, using reference counts to free up memory when some elements are no longer in use.

7.8.3 Filesystem bottleneck

Our experiments (presented in Chapter 8) show that a bottleneck exists at the filesystem level, slowing down our reference monitor. This is due to the frequency of updates on the extended attributes of files, which are performed in a very synchronized way every time a `read` or `write` access occurs. Recall from Chapter 4 that the extended attributes are represented as contiguous "flat" portions of memory. Therefore, accessing such information requires conversions to our in-memory representation of *information tags* on `read` access, and the other way round on `write` access. Furthermore, on `write` access, it is also required to load the *policy tags* of the files into

[4] Many thanks to Paul Moore for his kind help and cooperation.

[5] This patch was released on the mailing lists of the kernel, with the following subject: "cipso: handle CIPSO options correctly when NetLabel is disabled".

memory before checking the legality of their new content. A solution to cope with this shortcoming would be to maintain a cache of open file descriptors, containing for each file:

- The current policy tag.
- The current information tag.

7.8.4 Enforcement mode

Our primary goal is intrusion detection, therefore we do not block any information flow in the present model and implementation (we run in so-called *permissive* mode). However, the ability to enforce a policy may be considered in some situations including the deployment of an IPS (Intrusion Prevention System) based on our model, or setting up information flow control[6] in a trusted computing environment. As our implementation uses the LSM framework, providing access control mechanisms to security modules, the choice of enforcing the policy instead of raising an alert requires minor code modifications. Also, in terms of data structures, blocking illegal information flows reduces the amount of space required by the tainting: when running in *permissive* mode, we need to taint all the information present in all information flows. When enforcing a policy, some information flows are blocked, thus reducing the amount of tainting. Some simplifications can be done in the information tags in such a situation. Consider a container c, with a policy tag $ptag(c)$ and an information tag $itag(c)$. Recall the *Legal* relation from definition 3 in Chapter 2. When enforcing the policy, the state of the information tag of the container is always legal with respect to its policy tag: $Legal(itag(c), ptac(c))$ always stands. In such a case, the information tag of any container is always a subset of its policy tag.

One possible optimization of our implementation, when used in enforcement mode, would be the use of fixed-sized bitmaps to represent information tags, rather than doubly linked lists. The latter are very efficient in the case of dynamic allocation, when no size boundary exists. However, in the present case, the size of information tags is bound by the policy: each subset of the policy defines one possible legal state of the information tag of the container. For any policy tag $P = \{\{p_1\}, \{p_2\}, \ldots, \{p_N\}\}$, the corresponding legal information tag is bound by:

$$I = \bigcup_{1}^{N} p_i$$

We could represent such information tags in a fixed-size bitmap for every supervised container of the operating system, thus reducing the memory overhead of our implementation when enforcing the policy.

7.9 Conclusion

In this chapter, we have presented the distributed aspects of our implementation, relying on the Netlabel subsystem to attach CIPSO labels to network packets leaving each host of a group. The distributed token protocol (and the computation of *deltas*) has not been implemented at the moment. Instead, we use fixed-size bitmaps in the labels that we attach to network packets. In its current state, this implementation allowed us to perform the experiment presented in this thesis, and available for download from our website[7], released under the GPLv2 license. At the time of this writing, researchers outside our team have contacted us and started using it for other purposes, as a framework for information tainting, claiming that this is the only freely available implementation of such a tainting framework today. In an effort to distribute and cooperate on this project even more, our research team[8] is currently pursuing this project with several Ph.D. students and a research engineer.

[6] As opposed to information flow tracking, information flow control systems block illegal traffic.
[7] http://www.blare-ids.org
[8] The CIDre team, at Supélec, *www.supelec.fr*.

Chapter 8

Experiments

To conclude on the third part of this thesis, introducing network and distributed aspects to the intrusion detection model presented earlier in previous chapters, we will now detail our experiments based on the implementation explained in Chapter 7. We first present a case of intrusion on the client side, by visiting a malicious service using a web browser and a flawed plugin. The malicious service targets sensitive data on the client by using a remote exploit on the Java Virtual Machine. This first experiment shows how we are able to detect confidentiality violations and data leaks with our IDS along with a *network policy*, as introduced in Chapter 5.2. After this, we present a second experiment, involving a distributed web service composed of supervised hosts sharing a distributed information flow policy. We show how the reference monitor of each host is able to individually identify illegal information flows spawned by a successful attack. We finally present an assessment of the performances at the host level and discuss about the usability, advantages and shortcomings of our approach.

8.1 Data leaks through a web browser

This first experiment makes use of the *network policy*, that we introduced in Chapter 5.2. The following scenario, as illustrated on Figure 8.1 shows how our new model and implementation can detect confidentiality violations by untrusted code interpreted by a Web browser. Web browsers were initially simple applications displaying HTML content to the final user, but those have evolved into complex applications running JavaScript and other interpreted languages on the client machine, inevitably exposing user data to a number of real threats. In this scenario, a client is running a modified Linux kernel with our reference monitor, including the network extension that we presented in Chapter 5.2. The client visits a malicious web page using Mozilla Firefox 3.5 and the Java runtime environment plugin (JRE) version 6 update 10. This version is subject to the "Java calendar deserialization" vulnerability (CVE 2008-5353) that may lead to the execution of arbitrary code by an attacker. The client executes malicious Java code exploiting this issue and embedding a payload that allows the attacker to get a remote shell on the machine.

Assume the folder`/home/alice/confidential/` contains 64 confidential files. We labeled these files as being confidential, and assigned an *information tag* containing a unique identifier between 1 and 64 to each of them. The *information tag* of these files is a set containing one unique identifier, *e.g.,* `{1}`. This experiment is similar to the use case "all sensitive data must stay local" introduced in Chapter 5.3.1. We defined an empty *network policy tag* as follows :

$$P_{net} = \{\{\}\} = \bot$$

In this configuration, any application sending any of the labelled files to any remote host is a security policy violation and triggers an alert. Now we visited a crafted web page *http://www.malicious-host/malicious-page.html* embedding a malicious Java applet containing an attack against the previously mentioned vulnerability. This malicious page causes Mozilla Firefox

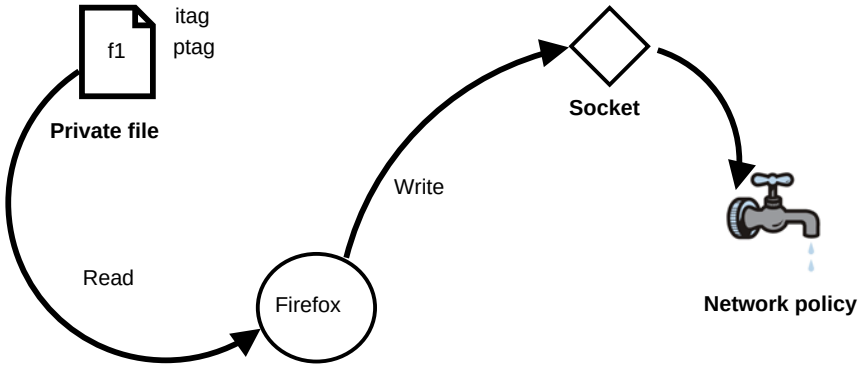

Figure 8.1: Monitoring outgoing information

to execute the Java virtual machine (JVM) in a separated process, which in turn interprets the Java code containing a remote shell allowing the attacker to connect to the local machine. As the attacker accesses labelled files of the local filesystem, the *information tag* of the process running Java is updated with *information tags* of the files it reads. At the moment when it sends information through a socket, our kernel reference monitor considers that the data being sent contains information from the files it previously read, and proceeds to a lookup throughout the *network policy tag* to ensure this behavior is allowed by the user. For every illegal attempt to illegally send information by the Java process, we were warned by the reference monitor with the following message:

```
[BLARE_POLICY_VIOLATION] Illegal information sent to socket by
process [PID] running java
```

8.2 Attack on a distributed web service

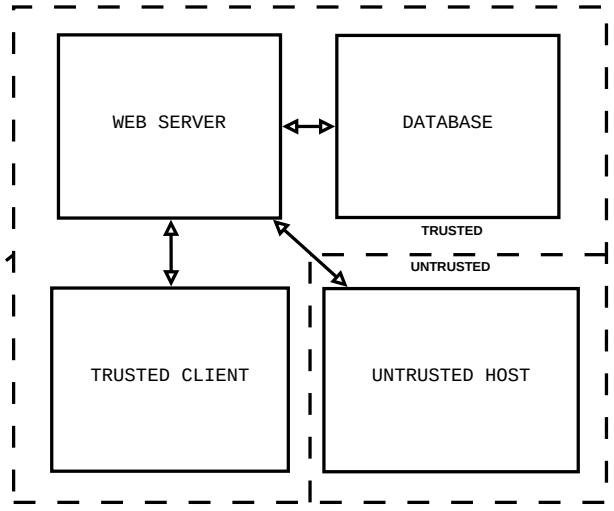

Figure 8.2: group of trusted hosts

The following describes an experiment in a distributed system. We have set up an attack scenario targeting a *group* of trusted hosts running our modified kernel. This *group* is composed of three hosts: a web server, a database server and a client, all three connected to the same Virtual Private Network (VPN). The web server (Apache) hosts two websites, isolated in two virtual hosts

www_1 and www_2 (Apache vhosts[1].) The database server (PostgreSQL) hosts two databases, storing data of the two virtual hosts: db_1 stores information related to www_1, and db_2 stores information related to www_2. Connections to www_1 are allowed from the outside. Connections to the other hosts of the VPN and to www_2 are forbidden from the outside. This policy is enforced by classical firewall rules. The following shows how it is possible with our intrusion detection model to detect illegal information flows between hosts caused by an intrusion. We used Debian Squeeze virtual guests running as KVM [29] instances. The two websites run Wordpress. The website www_1 runs the e-commerce plugin Foxypress[2]. We used the version 0.4.2.2 of this plugin, which is vulnerable to an upload exploit (EDB-ID: 18991)[3]. This vulnerability allows for arbitrary file upload and remote code execution.

8.2.1 Scenario

As shown on Figure 8.3, we labeled all the files of www_1 and www_2 as well as the PHP5 dynamic library (used by apache to interpret PHP code) with distinct *information tags* on the web server. On the database server, we labeled the PostgreSQL binary as well as two tables on each database. We could label information at the table level by using the option `default_with_oids = on` in PostgreSQL's configuration file. Object identifiers (OIDs) are used in PostgreSQL as primary keys for system tables, as well as user-created tables when using this option. Each table in PostgreSQL is mapped to a file named after its OID. Thus, we could label the files related to the supervised tables.

Host	Files	*Itag*	Execution
Web server	www_1	i_1	x_1
	www_2	i_2	x_2
	`libphp5.so`	i_{php}	x_{php}
	`apache2`	i_a	x_a
Database server	db_1: `wp_users`	i_{u_1}	x_{u_1}
	db_1: `wp_posts`	i_{p_1}	x_{p_1}
	db_2: `wp_users`	i_{u_2}	x_{u_2}
	db_2: `wp_posts`	i_{p_2}	x_{p_2}
	`postgres`	i_{pg}	x_{pg}

Figure 8.3: Labels on files

By default, both Apache and PostgreSQL create a new process for each connection. Recall the Run function from Definition 3.4.3. When a process executes a binary file (or the content of a dynamic library) labeled with i_k, its *information tag* is set to $x_k = Run(i_k)$. Therefore, both Apache and PostgreSQL processes always have their *information tags* initialized to respectively $x_a = Run(i_a)$ and $x_{pg} = Run(i_{pg})$. We used the following *policy tag* for both Apache and PostgreSQL processes: $P = \{\{x_a, x_{pg}, x_{php}, i_1, i_{u_1}, i_{p_1}\}\{x_a, x_{pg}, x_{php}, i_2, i_{u_2}, i_{p_2}\}\}$. Such a policy makes it illegal for any process running Apache or PostgreSQL to hold information from both websites simultaneously, or to run any code other than Apache and PosgreSQL binaries and libphp5. When an external visitor visits www_1, the web server creates a new process for this connection and reads files labeled with i_1. It also maps `libphp5.so` in executable memory pages which taints the process with x_{php}. It queries the database server. The database server forks a new process and reads information from db_1. At this stage, the *information tag* of the PostgreSQL process is tainted with $S_1 = \{x_a, x_{pg}, x_{php}, i_1, i_{u_1}, i_{p_1}\}$. After the PostgreSQL process has responded to the Apache

[1] From `http://www.apache.org`: the term Virtual Host refers to the practice of running more than one web site on a single machine.

[2] `www.foxy-press.com`

[3] `http://www.exploit-db.com/exploits/19100/`

process, both processes have equal *information tags*[4], as each process labels network packets with a CIPSO option containing its *information tag* (in a bitmap, as described in Chapter 7). When an internal host connects to the internal virtual host www_2, similar interactions happen between the hosts, and the *information tags* of both processes handling the connection are tainted with $S_2 = \{x_a, x_{pg}, x_{php}, i_2, i_{u_2}, i_{p_2}\}$. In both cases, information flows are legal, and so no alert is raised, because *information tags* are subsets of the *policy tags* in both containers: $S_1 \sqsubseteq P \wedge S_2 \sqsubseteq P$.

8.2.2 Attack

The following attack leaks information from the private web site www_2 located on the intranet. The attacker runs the upload exploit on the Foxypress plugin on www_1 and injects a malicious PHP file on the web server. We used Metasploit[5] to run the attack. After injecting the file, the running web server process's *information tag* was equal to $S_1 = \{x_a, x_{pg}, x_{php}, i_1, i_{u_1}, i_{p_1}\}$, and so was the *information tag* of the malicious PHP file. From there, any illegal action triggered an alert:

- Executing the malicious PHP file, which taints[6] the process's *information tag* with $Run(S_1) = \{x_1, x_{u_1}, x_{p_1}\}$ is illegal, as $Run(S_1) \not\sqsubseteq P$
- Querying the database server to access data from www_2, which taints the process's *information tag* with $S_3 = \{x_a, x_{pg}, x_{php}, i_1, i_{u_1}, i_{p_1}, i_{u_2}, i_{p_2}\}$ is illegal as well, as $S_3 \not\sqsubseteq P$.

Information tags are carried over the network through CIPSO labels, therefore both the web server and the database server raise an alert in the case of illegal information flow, as both servers are affected by the attack: data from the database server leaks, and the web server runs arbitrary code.

8.3 Evaluation of performances

The following is an evaluation of our implementation in terms of performances. In order to assess the performance overhead of our LSM module, we uncompressed a Linux kernel source tree and used it as a dataset containing 39048 files, that we individually labeled with a unique *information tag*. The machine we used is a Pentium 4 3.0 Ghz with 2.5 GB of RAM. We evaluated the performances of our kernel by transferring all the files of our dataset through a SSH tunnel, following the scenario "all sensitive data must stay local" as presented in Chapter 5.2.

Figure 8.4 compares the CPU idle time when using Linus Torvald's kernel (that we call Vanilla) and the Blare kernel. As expected, the Vanilla kernel gives lower CPU overhead during the transfer (higher CPU idle value). Our security framework adds 30% to 40% of extra overhead to the data transfer.

Figure 8.5 compares the memory overhead of our kernel and makes a comparison with a Vanilla kernel executing the same file transfer operation. As KBlare is attaching meta-information to every system object, the memory consumption remains higher by 30% on average when using our Kernel.

8.3.1 Overall completion time

The overall completion time was 300% longer with our kernel than with the Vanilla kernel. This limitation is due to a bottleneck at the filesystem level in our prototype (as described in Chapter 7). The extended attributes of the filesystem are used extensively in our implementation with no optimization. We believe that the overall performances of our system can be improved dramatically by optimizing the current prototype as follows: rather than updating *tags* at each filesystem operation (*i.e.* `fread` and `fwrite`), we could instead maintain a cache for *open* file descriptors, and synchronize it with the actual filesystem whenever a call to `fclose` is performed. An efficient cache may be implemented with a binary tree indexed on the inode numbers of each file.

[4] At the time of this experiment, we did not discard elements of $\mathcal{X}$ when receiving network packets, and the *network policy* and the policy of the process were combined into the policy tag of the process. This experiment is still valid in the current model, with minor changes in the way tags propagate.

[5] `http://www.metasploit.com`

[6] Apache maps PHP files in executable memory pages (`PROT_EXEC`), like it does with dynamic libraries.

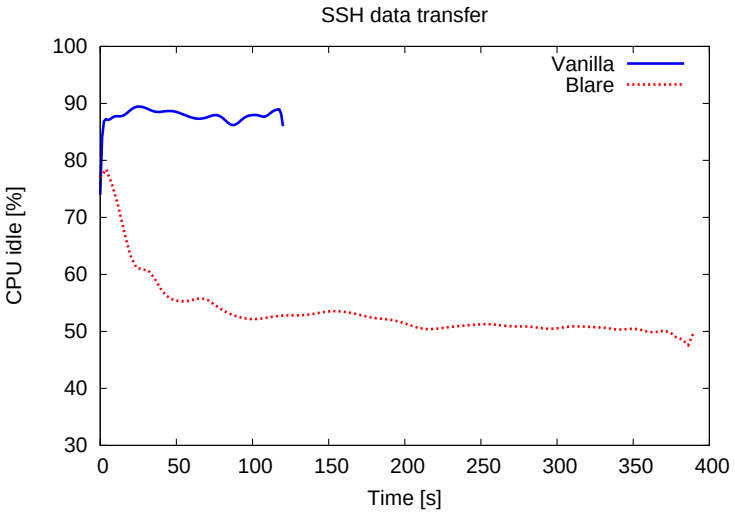

Figure 8.4: CPU overhead on SSH transfer

8.4 Discussion

8.4.1 Detection rate

When evaluating an intrusion detection system, a common measure is the rate of *false negatives* and *false positives*. By design, our conservative approach does not allow *false negatives*[7]. Our model of information flow tainting makes an overestimate of all possible content residing in containers, and maintains it updated after every information flow, both at the operating system level and on the network. Network traffic, or other forms of datasets, are a common basis for evaluating misuse IDSes, or anomaly IDSes based on statistical models. As our approach does not rely on network traffic analysis, nor on learned profiles, no such dataset can be used to evaluate our model. In our case, the dataset is determined by the pool of attacks we run. These attacks are included in the Metasploit framework as well as in the "Common Vulnerabilities and exposures" (CVE) database[8].

In our experiments, we have been able to successfully detect intrusions with no *false positives* as long as the system was following a legal behavior. Each time an event involving an illegal information flow occurred, all the subsequent information flows performed by the same process (or set of processes involved in the attack) in `read` access were considered illegal, as well as all the information flows towards supervised objects in `write` access (*i.e.* objects protected by a *policy tag* restraining their legal content).

Our model does not rely on a fixed information flow policy. The policy is manually adjusted to fit the different requirements of each supervised system. Therefore, the rate of *false positives* is highly variable. It depends on the following parameters:

1. The accuracy and consistency of the defined information flow policy.
2. The lifetime of tainted processes (these tend to accumulate more tags with time, leading to more *false positives*).
3. The use of IPC (Inter Process Communication).
4. The number of processes or services accessing the same set of files (including temporary files) or common objects.

It is impractical to perform a comprehensive study of the false positive rate in our current work. However, we can identify the following behavior from our experiments.

[7] Except in the case of eventual covert channels, which by nature are very difficult if not impossible to track. Furthermore, attacks relying on such methods are very uncommon.

[8] `http://cve.mitre.org/`

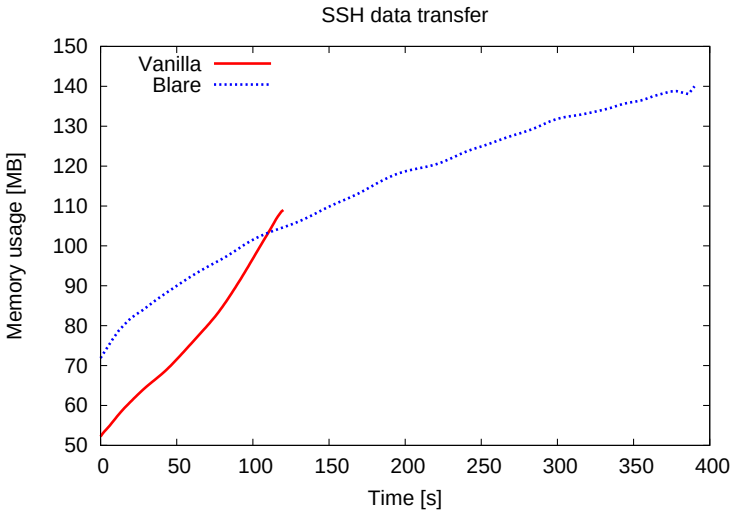

Figure 8.5: Memory overhead on SSH transfer

Situation with low false positive rates

Server-side services and applications often involve multiple processes, where each process handles one client connection, like in the experiment on a distributed web service previously presented in Section 8.2. Running the attack on the web server spawns illegal information flaws that we are able to detect. As both Apache and PostgreSQL fork one process per connection, the number of alerts reported by our reference monitor *after* an illegal information flow occurs is limited by the lifetime of such processes. Once a connection ends, the related process is killed. When a new connection occurs, a new process is forked from a "clean" process: the so-called *worker* process, from which new processes are forked, does not get tainted by information flows of its child processes, and every new connection leads to an untainted process. Furthermore, such server-side applications handle isolated sets of files (*e.g.* Apache works with files in `/var/www` where PostgreSQL stores its database tables in files located in `/var/lib/postgresql`), which eases the task of defining suitable information flow policies.

Situations with high false positive rates

On the contrary, desktop applications often involve buses such as DBUS, graphical environments, and other long term processes, staying active until the current user closes his or her session. Defining a suitable information flow policy in such a situation is more complex. Furthermore, by computing an overestimate of possible information flows, our reference monitor lacks accuracy in this context.

Recall the experiment from Section 8.1. In this scenario, alerts are reported when sensitive data may have left the system through a network connection. When conducting this experiment, no false positives occurred until the web browser accessed sensitive information. From this point on, all subsequent information flows were considered illegal. This approach is valid for tracking access to sensitive information which should by no mean leave the system, and where access is performed by unwanted and/or malicious events.

In other situations, where a finer analysis inside applications' code is required, our approach involves a high number of false positives, and lacks accuracy.

8.4.2 Improving accuracy

In this Ph.D., we focused on OS-level mechanisms. Our model and implementation provide a basis for system-wide intrusion detection based on taint marking. The level of granularity of our approach in terms of tracking is limited, in our current implementation, by the abstraction of UNIX systems. Figure 8.6 illustrates our approach. It represents a process with inputs i_1, i_2, i_3

and outputs o_1, o_2, o_3. From our level of abstraction, we cannot determine how information flows within processes (or applications). Therefore, we compute an overestimate of the possible flows: the outputs of the process are considered as function of all the previous inputs, at any time. This overestimate generates variable amounts of *false positives* depending on the context, as presented in the previous section.

Taking this current work as a basis, a solution to dramatically reduce the amount of false positives is to increase the accuracy of our data flow analysis. By combining application-level information flow tracking techniques with our OS-level reference monitor, it becomes feasible to finely observe information flows within processes, and to supervise multiple applications as well as their interactions through the operating system in a fine-grained manner. This aspect is out of the scope of this Ph.D., and is part of current research in the CIDre team.

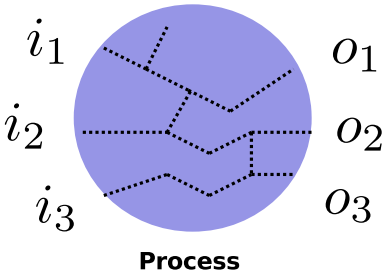

Figure 8.6: Information flows within applications

8.4.3 Usability

This model does not replace access control mechanisms, nor enforce any security policy but instead helps to ensure that no unwanted behaviour happens with respect to defined sets of information managed by users and applications of multiple hosts. The situation where a web-browser accesses some personal information is a good example of our goals: where access control mechanisms could have been used to block this particular access in the first place, it would not prevent applications from indirectly accessing the same information by another channel (shared memory, IPC with another application etc.). Furthermore, in this example, we focus on the fact that this information should not leave the system through the network, therefore no alert would be raised when an application accesses such information but does not send it over.

The main limitation of our OS-level approach is its accuracy, as it computes an over-estimate of the actual content involved in information flows. This has a direct impact on the *false positives* rate. Work in progress in the CIDre team seeks to address this shortcoming by several means:

1. By filtering alerts in userspace. For instance any sequence of false positives triggered by the same event can safely be discarded after the event has been reported.

2. By using our framework in cooperation with language or architecture-level monitors.

3. By instrumenting native applications.

8.5 Conclusion

In this chapter, we presented our experiments, as well as an analysis of the performances and accuracy of our intrusion detection model. We have shown that it is suitable for detecting intrusions in both isolated and distributed systems. The maximum performance penalty that we have measured was close to 30% in terms of memory overhead, and 30-40% in terms of CPU overhead. Due to a bottleneck at the filesystem level in our current implementation, the overall completion time of our experiments was 300% longer when using our IDS. We believe that this could be dramatically improved by the mean of optimizations (*e.g.,* using caches). We have identified situations where our

model is suitable for realistic intrusion detection, as well as situations highlighting its shortcomings in terms of accuracy, leading to high rates of *false positives*. Solutions exist so as to address the identified shortcomings, and are considered in current research in the CIDre project team.

Conclusion

In response to the complexity of securing ever evolving information systems, often relying on distributed services across multiple hosts, we have designed and implemented an information flow model using taint marking techniques, in order to detect intrusions at the OS kernel level. Our approach of *anomaly detection* is based on the specification of an information flow policy. By tracking information flows between objects such as *files, sockets, pipes, memory mappings etc.*, as well as in network packets flowing between hosts, we are able to successfully detect intrusions, both in isolated hosts and in distributed services composed of multiple hosts (gathered in *groups*).

We have presented our model of information flow tracking, specifying a fine-grained policy at four different levels: containers of information, users, applications and network. Our reference monitor was implemented in the Linux kernel, as a Linux Security Module. This model and its implementation represent our first contribution. The validation of the implementation was experimental. For each experiment, the involved aspects in the theoretical model were identified, and the results were compared to the expected behavior of the system with respect to the theory. Our new intrusion detection principles have been validated through our experiments. In Chapter 8, we have practically set up and presented two realistic applications of this approach. A first application followed a scenario involving an attack against confidentiality, by exploiting a security flaw in a plugin, inside a web browser. We demonstrated that our model was able to successfully detect the illegal information flaws spawned by the attack. A second application focused on distributed services across several hosts. Our reference monitor was successful at detecting attacks against a frontal web server. Illegal information flaws spawned by the attacked web server, communicating with remote processes, were also detected at the level of each host composing the distributed service, and alerts were reported by each reference monitor. The extension of our model and implementation to distributed systems represents the second contribution of this work.

The performance overhead of our reference monitor reaches 30% in memory consumption, and 30-40% in CPU, in extreme situations involving a high number of distinct taint information. Its main limitation is an overhead on the completion time of some operations in some cases, reaching up to 300% in extreme situations. Our current prototype may be further optimized so as to decrease the involved performance penalty, and we proposed possible tracks for improvement in Chapter 7.

Our model and its implementation are suitable for the following applications:

- Supervision of users and programs: our model can be used to track applications by attaching a policy (*i.e.*, a set of policy rules) to their related code (binary programs, scripts, shared libraries *etc.*). A policy may also be attached to local users. When a process executes some code, such a policy is used along with the policy of the current user (if defined) to determine the legal information flows caused by the resulting processes. Any violation of the policy triggers an alert. This may be used to protect users' privacy, as well as the integrity of information.

- Supervision of network communications: a *network policy* can be used to define the legal interactions between processes (*i.e.,* applications executed by users) of different hosts involving sets of supervised information.

- Tracking the changes made by viruses: by keeping the origins of all data present in each container, we can retrieve all the information flows that were caused by a virus (or any given

piece of executable information). This may be used *e.g.* to and track the modifications that were made in order to perform a *rollback* of the system to a safe state.

- Detect the presence of an attacker by detecting abnormal behavior of programs, services or daemons.
- Detect the execution of modified applications and rootkits: as we do not trust code that has been illegally modified, we can detect malware and rootkits. When the code of an application or library is altered by a process, we keep tracks of such changes in the *information tag* of the modified application file(s). These *meta-data* give information about the running code as well as information hold by the process which altered the file. Whenever such changes are illegal, the execution of the new code is illegal too.

The framework that we presented provides a basis for system-wide intrusion detection in distributed systems and services. The overall accuracy of our model depends on the level of granularity offered by the underlying OS abstractions. Even though we were able to successfully detect intrusions with this model, it presents *shortcomings* in situations where accuracy is required, as shown in Chapter 8. It is impractical, at the OS level, to finely observe information flows *within* applications. Therefore, in its current state, our framework is usable in simple situations, but it generates high rates of *false positives* in environments where processes communicate with IPC mechanisms. In order to address these shortcomings, current work in the CIDre team focuses on the cooperation of our OS-level reference monitor with application-level reference monitors.

Our model may also be further distributed in future work. We proposed a distributed protocol allowing hosts of a *group* to exchange security tokens in a peer to peer fashion. While the resolvers on each host manage information tainting in a fully distributed manner, the specification of the policy in our current work is done manually on each host of the *groups*, by a central system administrator. The specification of the policy could instead be determined independently on each host in a decentralized way. Such a policy could rely on a peer-to-peer protocol, allowing each pair of hosts to agree on a common set of rules, regarding legal interactions of their processes with respect to the data they manage.

Bibliography

[1] T. AbuHmed, A. Mohaisen, and D.H. Nyang. A survey on deep packet inspection for intrusion detection systems. *Arxiv preprint arXiv:0803.0037*, 2008.

[2] Ross J. Anderson, Frank Stajano, and Jong-Hyeon Lee. Security policies. *Advances in Computers*, 55:186–237, 2001.

[3] Stefan Axelsson. Intrusion detection systems: A taxonomy and survey. Technical Report 99-15, Dept. of Computer Engineering, Chalmers University of Technology, March 2000.

[4] D. E. Bell and L. J. LaPadula. Secure computer system: Unified exposition and multics interpretation. Mtr-2997 (esd-tr-75-306), MITRE Corp., 1976.

[5] Massimo Bernaschi, Emanuele Gabrielli, and Luigi V. Mancini. Remus: a security-enhanced operating system. *ACM Trans. Inf. Syst. Secur.*, 5:36–61, February 2002.

[6] D. F. C. Bewer and M. J. Nash. The chinese wall security policy. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1989.

[7] K. Biba. Integrity considerations for secure computer systems. Technical Report N ESD-TR 76-372, MITRE Co., April 1977.

[8] Jeff Bonwick and Sun Microsystems. The slab allocator: An object-caching kernel memory allocator. In *In USENIX Summer*, pages 87–98, 1994.

[9] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[10] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. In *Internet Mathematics*, pages 636–646, 2002.

[11] Suresh N. Chari and Pau-Chen Cheng. Bluebox: A policy-driven, host-based intrusion detection system. *ACM Trans. Inf. Syst. Secur.*, 6:173–200, May 2003.

[12] Yi-Ming Chen and Yung-Wei Kao. Information flow query and verification for security policy of security-enhanced linux. In *Proceedings of IWSEC*, pages 389–404, 2006.

[13] Winnie Cheng, Dan R. K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. Abstractions for usable information flow control in aeolus. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 12–12, Berkeley, CA, USA, 2012. USENIX Association.

[14] D. D. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the IEEE Symposium on Security and Privacy (SSP'87)*, pages 184–194. IEEE Society Press, mai 1987.

[15] Secure Computing Corporation. Dtos general system security and assurability assessment report. Technical report, Secure Computing Corporation, 1997.

[16] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a taxonomy of intrusion-detection systems. *Computer Networks*, 31(8):805–822, 1999.

[17] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976.

[18] Dorothy E. Denning. An Intrusion-Detection Model. *IEEE transaction on Software Engineering*, 13(2):222–232, 1987.

[19] Department of Defense. Trusted network interpretation of the DoD TCSEC (red book). NCSC-TG-005, 1987.

[20] Ulrich Drepper. How to write shared libraries. Technical report, Red Hat, Inc., 2011.

[21] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the asbestos operating system. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 17–30, New York, NY, USA, 2005. ACM.

[22] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, 2010.

[23] Simon N. Foley, Stefano Bistarelli, Barry O'Sullivan, John Herbert, and Garret Swart. Multilevel security and the quality of protection. In *Proceedings of First Workshop on Quality of Protection*, page 2006. Springer, 2006.

[24] Pedro Garcia-Teodoro, Jesús E. Díaz-Verdejo, Gabriel Maciá-Fernández, and E. Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1-2):18–28, 2009.

[25] Carrie Gates and Carol Taylor. Challenging the anomaly detection paradigm: a provocative discussion. In *Proceedings of the 2006 workshop on New security paradigms*, NSPW '06, pages 21–29, New York, NY, USA, 2007. ACM.

[26] Laurent Georges, Valérie Viet Triem Tong, and Ludovic Mé. Blare tools: A policy-based intrusion detection system automatically set by the security policy. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID 2009)*, 2009.

[27] J. Goguen and J. Meseguer. Unwinding and inference control. In *IEEE Symposium on Security and Privacy*, 1984.

[28] Irfan Habib. Getting started with the linux intrusion detection system. *Linux J.*, 2006, March 2006.

[29] Irfan Habib. Virtualization with kvm. *Linux J.*, 2008(166), February 2008.

[30] Toshiharu Harada, Takashi Horie, and Kazuo Tanaka. Access policy generation system based on process execution history. *Network Security Forum*, 2003.

[31] Christophe Hauser, Frédéric Tronel, Colin Fidge, and Ludovic Mé. Distributed intrusion detection, an approach based on taint marking. *Proceedings of the IEEE International Conference on Computer Communications (ICC)*, 2013.

[32] Tronel F. Reid J. Hauser, C. and C. Fidge. A taint marking approach to confidentiality violation detection. In C. Pieprzyk, J.and Thomborson, editor, *Australasian Information Security Conference (AISC 2012)*, volume 125 of *CRPIT*, pages 83–90, Melbourne, Australia, 2012. ACS.

[33] Alejandro Hernandez and Flemming Nielson. Enforcing mandatory access control in distributed systems using aspect orientation. In *21st Nordic Workshop on Programming Theory – NWPT2009*, pages 62–64, 2009.

[34] G. Hiet, L. Mé, B. Morin, and V. Viet Triem Tong. Monitoring both os and program level information flows to detect intrusions against network servers. In *IEEE Workshop on "Monitoring, Attack Detection and Mitigation"*, 2007.

[35] Kenneth Ingham and Stephanie Forrest. A History and Survey of Network Firewalls. Technical report, 2002.

[36] Hajime Inoue and Stephanie Forrest. Anomaly intrusion detection in dynamic execution environments. In *Proceedings of the 2002 workshop on New security paradigms*, NSPW '02, pages 52–60, New York, NY, USA, 2002. ACM.

[37] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Network and Distributed Systems Security Symposium*, 1999.

[38] Amy L. Herzog Joshua D. Guttman and John D. Ramsdell. Information flow in operating systems : Eager formal methods. *Workshop on Issues on the Theory of Security (WITS)*, 2003.

[39] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proceedings of 9th USENIX Security Symposium (SEC 2000)*, August 2000.

[40] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. In *Proceedings of the 21st Symposium on Operating Systems Principles*, Stevenson, WA, October 2007.

[41] Butler W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8:18–24, January 1974.

[42] Leonard J. LaPadula and D. Elliott Bell. Secure computer systems: A mathematical model. MTR-2547 (ESD-TR-73-278-II) Vol. 2, MITRE Corp., Bedford, may 1973.

[43] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *Proceedings of the 21st National Information Systems Security Conference*, pages 303–314, 1998.

[44] R. Love. *Linux Kernel Development.* Novell Press. Novell Press, 2005.

[45] Claudio Mazzariello, Roberto Bifulco, and Roberto Canonico. Integrating a network ids into an open source cloud computing environment. In *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, pages 265–270. IEEE, 2010.

[46] Jonathan M. McCune, Trent Jaeger, Stefan Berger, Ramón Cáceres, and Reiner Sailer. Shamon: A system for distributed mandatory access control. In *Proceedings of ACSAC*, pages 23–32, 2006.

[47] Paul E. McKenney and Jonathan Walpole. Introducing technology into the Linux kernel: a case study. *SIGOPS Oper. Syst. Rev.*, 42(5):4–17, 2008.

[48] K.W. Miller, J. Voas, and G.F. Hurlburt. Byod: Security and privacy considerations. *IT Professional*, 14(5):53–55, 2012.

[49] Mark Miller, Ka-Ping Yee, Jonathan Shapiro, and Combex Inc. Capability myths demolished. Technical report, 2003.

[50] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31(5):129–142, 1997.

[51] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol.*, 9(4):410–442, 2000.

[52] Pablo Neira-Ayuso, Rafael M. Gasca, and Laurent Lefevre. Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience*, 40(9):797–810, 2010.

[53] James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2005)*, 2005.

[54] Novell/SUSE. Apparmor, application security for linux. Technical report.

[55] Department of Defense. Trusted computer system evaluation criteria (orange book). DoD 5200.28-STD, 1983.

[56] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Netw.*, 51:3448–3470, August 2007.

[57] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proc. of the 7th Usenix Security Symposium*, pages 31–51, San Antonio, TX, January 1998.

[58] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, April 2006.

[59] Martin Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX LISA'99 conference*, pages 229–238, Seattle, WA, November 1999.

[60] S. Roschke, Feng Cheng, and C. Meinel. Intrusion detection in the cloud. In *Dependable, Autonomic and Secure Computing, 2009. DASC 09. Eighth IEEE International Conference on*, pages 729–734, 2009.

[61] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, 2009.

[62] Ravi S. Sandhu. Lattice-based access control models. *Computer*, 26(11):9–19, 1993.

[63] Casey Schaufler. The simplified mandatory access control kernel. Technical report.

[64] Chris Vance Stephen Smalley. Implementing SELinux as a Linux Security Module. Technical report, NAI Labs, 2002.

[65] Stéphane Geller, Christophe Hauser, Frédéric Tronel, Valérie Viet Triem Tong. Information flow control for intrusion detection derived from mac policy. *Proceedings of the IEEE International Conference on Computer Communications (ICC)*, 2011.

[66] Valérie Viet Triem Tong, Andrew Clark, and Ludovic Mé. Specifying and enforcing a fine-grained information flow policy: Model and experiments. In *Proceedings of MIST*, 2010.

[67] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[68] Giovanni Vigna, William Robertson, and Davide Balzarotti. Testing network-based intrusion detection signatures using mutant exploits. In *The ACM Conference on Computer and Communication Security (ACM CCS*, pages 21–30, 2004.

[69] David A. Wagner. Janus: an approach for confinement of untrusted applications. Technical report, Berkeley, CA, USA, 1999.

[70] Robert Watson and Chris Vance. The trustedbsd mac framework: Extensible kernel access control for freebsd 5.0. In *In USENIX Annual Technical Conference*, pages 285–296, 2003.

[71] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *USENIX Security Symposium*, pages 17–31, 2002.

[72] Ruoyu Wu, Gail-Joon Ahn, Hongxin Hu, and Mukesh Singhal. Information flow control in cloud computing. In *CollaborateCom'10*, pages 1–7, 2010.

[73] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 116–127, 2007.

[74] Mukarram Bin Tariq Yoges Mundada, Anirudh Ramachandran and Nick Feamster. Practical data-leak prevention for legacy applications in enterprise networks. Technical report, Georgia Institute of Technology, 2011.

[75] Krzysztof Zaraska. Prelude ids: current state and development perspectives, 2003.

[76] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

[77] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association.

[78] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware enforcement of application security policies using tagged memory. In Richard Draves and Robbert van Renesse, editors, *Proceedings of OSDI*, pages 225–240. USENIX Association.

[79] Xiaolan Zhang, Antony Edwards, and Trent Jaeger. Using cqual for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, Berkeley, CA, USA, 2002. USENIX Association.

[80] Hu Zhengbing, Li Zhitang, and Wu Junqi. A novel network intrusion detection system (nids) based on signatures search of data mining. In *Proceedings of the 1st international conference on Forensic applications and techniques in telecommunications, information, and multimedia and workshop*, e-Forensics '08, pages 45:1–45:7, 2008.

[81] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Introducing reference flow control for detecting intrusion symptoms at the os level. In Andreas Wespi, Giovanni Vigna, and Luca Deri, editors, *Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID'2002)*, volume 2516 of *Lecture Notes in Computer Science*, pages 292–306. Springer, 2002.

[82] Jacob Zimmermann, Ludovic Mé, and Christophe Bidan. Experimenting with a policy-based hids based on an information flow control model. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2003.

[83] Jacob Zimmermann and George Mohay. Distributed intrusion detection in clusters based on non-interference. In *Proceedings of the 2006 Australasian workshops on Grid computing and e-research - Volume 54*, ACSW Frontiers '06, pages 89–95, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

Appendix A

System Calls

The following is the list of all system calls on Linux-3.2/x86_64. We analyzed the semantics of all system calls in order to determine in which cases information flows may occur between two or more objects of the operating system. In our implementation, we track information flows resulting in a communication between userspace processes. We consider the kernel as trusted (if the attacker can modify the kernel, he already has full access over the system). Special cases, where information flows may occur, potential hidden channels may exist, particular aspects are highlighted, are numbered in the *information flow* column of the table below, and are described at the end of this section. The system calls that we track in our implementation are marked with a cross in the **Tracked** column. A $\sim$ symbol indicates that only a partial or indirect tracking is performed.

Number	Name	Description	Flow	Tracked
0	read	read from a file descriptor	yes[1]	✓
1	write	write to a file descriptor	yes[1]	✓
2	open	open and possibly create a file or device	no[1]	
3	close	close a file descriptor	no	
4	stat	get file status	no	
5	fstat	—	no	
6	lstat	—	no	
7	poll	wait for some event on a file descriptor	no	
8	lseek	reposition read/write file offset	no	
9	mmap	map files or devices into memory	yes	
10	mprotect	set protection on a region of memory	yes[2]	$\sim$
11	munmap	unmap files or devices into memory	no[3]	✓
12	brk	change data segment size	no	
13	rt_sigaction	examine and change a signal action	no	
14	rt_sigprocmask	examine and change blocked signals	no	
15	rt_sigreturn	return from signal handler and cleanup stack frame	no	
16	rt_ioctl	manipulates the underlying device parameters of special files	no[4]	

Number	Name	Description	Flow	Tracked
17	pread64	read from from a file descriptor at a given offset	yes[1]	✓
18	pwrite64	write to a file descriptor at a given offset	yes[1]	✓
19	readv	read data from multiple buffers	yes[1]	✓
20	writev	write data into multiple buffers	yes[1]	✓
21	access	check real user's permissions for a file	no	
22	pipe	create pipe	no[1]	
23	select	synchronous I/O multiplexing	no	
24	sched_yield	yield the processor	no	
25	mremap	remap a virtual memory address	no[5]	
26	msync	synchronize a file with a memory map	yes[6]	∼
27	mincore	determine whether pages are resident in memory	no	
28	madvise	give advice about use of memory	no	
29	shmget	allocates a shared memory segment	no[7]	
30	shmat	attaches the shared memory segment identified by shmid to the address space of the calling process	yes[7]	✓
31	shmctl	shared memory control	no	
32	dup	duplicate a file descriptor	no[1]	
33	dup2	—	no[1]	
34	pause	wait for signal	no	
35	nanosleep	high-resolution sleep	no	
36	getitimer	get value of an interval timer	no	
37	alarm	set an alarm clock for delivery of a signal	no	
38	setitimer	set value of an interval timer	no	
39	getpid	process identification	no	
40	sendfile	transfer data between file descriptors	yes[1]	✓
41	socket	create an endpoint for communication	no[8]	
42	connect	initiate a connection on a socket	no[8]	
43	accept	accept a connection on a socket	no[8]	
44	sendto	send a message on a socket	yes[8]	✓
45	recvfrom	receive a message from a socket	yes[8]	✓
46	sendmsg	send a message on a socket	yes[8]	✓
47	recvmsg	receive a message from a socket	yes[8]	✓
48	shutdown	shut down part of a full-duplex connection	no	

Number	Name	Description	Flow	Tracked
49	bind	bind a name to a socket	no[8]	
50	listen	listen for connections on a socket	no[8]	
51	getsockname	get socket name	no	
52	getpeername	get name of connected peer socket	no	
53	socketpair	create a pair of connected sockets	no[8]	
54	setsockopt	set options on sockets	no	
55	getsockopt	get options on sockets	no	
56	clone	create a child process	yes[9]	✓
57	fork	create a child process	yes[9]	✓
58	vfork	create a child process and block parent	—	
59	execve	execute program	yes[10]	✓
60	exit	terminate the calling process	no	
61	wait4	wait for process to change state, BSD style	no	
62	kill	send signal to a process	no	
63	uname	get name and information about current kernel	no	
64	semget	get a semaphore set identifier	no	
65	semop	semaphore operations	no	
66	semctl	semaphore control operations	no	
67	shmdt	detaches a shared memory segment	no[11]	
68	msgget	get a message queue identifier	no	
69	msgsnd	send message to a message queue	yes[12]	✓
70	msgrcv	receive message from a message queue	yes[12]	✓
71	msgctl	message control operations	no	
72	fcntl	manipulate file descriptor	no	
73	flock	apply or remove an advisory lock on an open file	no	
74	fsync	synchronize a file's in-core state with storage device	no	
75	fdatasync	—	—	
76	truncate	truncate a file to a specified length	no	
77	ftruncate	—	—	
78	getdents	get directory entries	no	
79	getcwd	Get current working directory	no	
80	chdir	change working directory	no	
81	fchdir	—	—	
82	rename	change the name or location of a file	no	
83	mkdir	create a directory	no	

Number	Name	Description	Flow	Tracked
84	rmdir	delete a directory	no	
85	creat	open and possibly create a file or device	no	
86	link	make a new name for a file	no	
87	unlink	delete a name and possibly the file it refers to	no	
88	symlink	make a new name for a file	no	
89	readlink	read value of a symbolic link	no	
90	chmod	change permissions of a file	no	
91	fchmod	—	—	
92	chown	change ownership of a file	no	
93	fchown	—	—	
94	lchown	—	—	
95	umask	set file mode creation mask	no	
96	gettimeofday	get time	no	
97	getrlimit	get resource limit	no	
98	getrusage	get resource usage	no	
99	sysinfo	returns information on overall system statistics	no	
100	times	get process times	no	
101	ptrace	process trace	yes[13]	
102	getuid	get user identity	no	
103	syslog	read and/or clear kernel message ring buffer; set console_loglevel	no	
104	getgid	get group identity	no	
105	setuid	set user identity	no	
106	setgid	set group id	no	
107	geteuid	get user identity	no	
108	getegid	get group id	no	
109	setpgid	set process group	no	
110	getppid	get process identification	no	
111	getpgrp	get process group	no	
112	setsid	creates a session and sets the process group ID	no	
113	setreuid	set real and/or effective user ID	no	
114	setregid	set real and/or effective group ID	no	
115	getgroups	get list of supplementary group IDs	no	
116	setgroups	set —	—	
117	setresuid	set real, effective and saved user ID	no	
118	getresuid	get real, effective and saved user IDs	no	

Number	Name	Description	Flow	Tracked
119	setresgid	set real, effective and saved group ID	no	
120	getresgid	get real, effective and saved group ID	no	
121	getpgid	get process group	no	
122	setfsuid	set user identity used for file system checks	no	
123	setfsgid	set group identity used for file system checks	no	
124	getsid	get session ID	no	
125	capget	get capabilities of thread(s)	no	
126	capset	set capabilities of thread(s)	no	
127	rt_sigpending	examine pending signals	no	
128	rt_sigtimedwait	synchronously wait for queued signals	no	
129	rt_sigqueueinfo	queue a signal and data to a process	no[14]	
130	rt_sigsuspend	wait for a signal	no	
131	sigaltstack	set and/or get signal stack context		
132	utime	change file last access and modification times	no	
133	mknod	create a special or ordinary file	no[15]	
134	uselib	load shared library	yes[16]	✓
135	personality	set the process execution domain	no	
136	ustat	get file system statistics	no	
137	statfs	—	—	
138	fstatfs	—	—	
139	sysfs	get file system type information	no	
141	getpriority	get program scheduling priority	no	
141	setpriority	set program scheduling priority	no	
142	sched_setparam	set scheduling parameters	no	
143	sched_getparam	get scheduling parameters	no	
144	sched_setscheduler	set scheduling policy/parameters	no	
145	sched_getscheduler	get scheduling policy/parameters	no	
146	sched_get_priority_max	get static priority range	no	
147	sched_get_priority_min	—	no	
148	sched_rr_get_interval	get the SCHED_RR interval for the named process	no	
149	mlock	lock memory	no	
150	munlock	unlock memory	no	
151	mlockall	local memory	no	
152	munlockall	unlock memory	no	
153	vhangup	virtually hangup the current tty	no	

Number	Name	Description	Flow	Tracked
154	modify_ldt	get or set ldt	no	
155	pivot_root	change the root file system	no	
156	_sysctl	read/write system parameters	no	
157	prctl	operations on a process	no	
158	arch_prctl	set architecture-specific thread state	no	
159	adjtimex	tune kernel clock	no	
160	setrlimit	set resource limits	no	
161	chroot	change root directory	no	
162	sync	commit buffer cache to disk	no	
163	acct	switch process accounting on or off	no	
164	settimeofday	set time	no	
165	mount	mount a filesystem	no	
166	umount2	umount a file system	no	
167	swapon	start swapping to file/device	yes[17]	
168	swapoff	stop swapping to file/device	no	
169	reboot	reboot or enable/disable Ctrl-Alt-Del	no	
170	sethostname	set hostname	no	
171	setdomainname	set domain name	no	
172	iopl	change I/O privilege level	no	
173	ioperm	set port input/output permissions	no	
174	create_module	create a loadable module entry	no	
175	init_module	initialize a loadable module entry	yes[18]	
176	delete_module	delete a loadable module entry	no	
177	get_kernel_syms	retrieve exported kernel and module symbols	no	
178	query_module	query the kernel for various bits pertaining to modules	no	
179	quotactl	manipulate disk quotas	no	
180	nfsservctl	syscall interface to kernel nfs daemon	no	
181	getpmsg	receive next message from a STREAMS file (not implemented)	yes	
182	putpmsg	send a message on a STREAM (not implemented)	yes	
183	afs_syscall	not implemented	n/a	
184	tuxcall	not implemented	n/a	
185	security	not implemented	n/a	
186	gettid	get thread identification	no	
187	readahead	perform file readahead into page cache	yes[1]	✓

Number	Name	Description	Flow	Tracked
188	setxattr	set an extended attribute value	yes[19]	
189	lsetxattr	—	—	
190	fsetxattr	—	—	
191	getxattr	retrieve an extended attribute value	yes[19]	
192	lgetxattr	—	—	
193	fgetxattr	—	—	
194	listxattr	list extended attribute names	no	
195	llistxattr	—	—	
196	flistxattr	—	—	
197	removexattr	remove an extended attribute	no	
198	lremovexattr	—	—	
199	fremovexattr	—	—	
200	tkill	send a signal to a thread	no	
201	time	get time in seconds	no	
202	futex	Fast Userspace Locking system call	no	
203	sched_setaffinity	set a process's CPU affinity mask	no	
204	sched_getaffinity	get a process's CPU affinity mask	no	
205	set_thread_area	Set a Thread Local Storage (TLS) area	no	
206	io_setup	create an asynchronous I/O context	no	
207	io_destroy	destroy an asynchronous I/O context	no	
208	io_getevents	read asynchronous I/O events from the completion queue	no	
209	io_submit	submit asynchronous I/O blocks for processing	no	
210	io_cancel	cancel an outstanding asynchronous I/O operation	no	
211	get_thread_area	Get a Thread Local Storage (TLS) area	no	
212	lookup_dcookie	return a directory entry's path	no	
213	epoll_create	open an epoll file descriptor	no	
214	epoll_ctl_old			
215	epoll_wait_old			
216	remap_file_pages	create a nonlinear file mapping	no	
217	getdents64	get directory entries	no	
218	set_tid_address	set pointer to thread ID	no	
219	restart_syscall	restart a system call	no	
220	semtimedop	semaphore operation	no	
221	fadvise64	predeclare an access pattern for file data	no	
222	timer_create	create a POSIX per-process timer	no	

Number	Name	Description	Flow	Tracked
223	timer_settime	arm/disarm and fetch state of POSIX per-process timer	no	
224	timer_gettime	—	no	
225	timer_getoverrun	get overrun count for a POSIX per-process timer	no	
226	timer_delete	delete a POSIX per-process timer	no	
227	clock_settime	clock and time functions	no	
228	clock_gettime	—	—	
229	clock_getres	—	—	
230	clock_nanosleep	high-resolution sleep with specifiable clock	no	
231	exit_group	exit all threads in a process	no	
232	epoll_wait	wait for an I/O event on an epoll file descriptor	no	
233	epoll_ctl	control interface for an epoll descriptor	no	
234	tgkill	send a signal to a thread	no	
235	utimes	change file last access and modification times	no	
236	vserver	not implemented	n/a	
237	mbind	Set memory policy for a memory range	no	
238	set_mempolicy	set default NUMA memory policy for a process and its children	no	
239	get_mempolicy	Retrieve NUMA memory policy for a process	no	
240	mq_open	open a message queue	no	
241	mq_unlink	remove a message queue	no	
242	mq_timedsend	send a message to a message queue	yes[12]	✓
243	mq_timedreceive	receive a message from a message queue	yes[12]	✓
244	mq_notify	register for notification when a message is available	no	
245	mq_getsetattr	get/set message queue attributes	no	
246	kexec_load	load a new kernel for later execution	yes[20]	
247	waitid	wait for process to change state	no	
248	add_key	Add a key to the kernel's key management facility	yes[21]	
249	request_key	Request a key from the kernel's key management facility	yes[21]	
250	keyctl	Manipulate the kernel's key management facility	no	
251	ioprio_set	set I/O scheduling class and priority	no	

Number	Name	Description	Flow	Tracked
252	ioprio_get	get I/O scheduling class and priority	no	
253	inotify_init	initialize an inotify instance	no	
254	inotify_add_watch	add a watch to an initialized inotify instance	no	
255	inotify_rm_watch	remove an existing watch from an inotify instance	no	
256	migrate_pages	move all pages in a process to another set of nodes	no	
257	openat	open a file relative to a directory file descriptor	no	
258	mkdirat	create a directory relative to a directory file descriptor	no	
259	mknodat	create a special or ordinary file relative to a directory file descriptor	no	
260	fchownat	change ownership of a file relative to a directory file descriptor	no	
261	futimesat	change timestamps of a file relative to a directory file descriptor	no	
262	newfstatat	get file status relative to a directory file descriptor	no	
263	unlinkat	remove a directory entry relative to a directory file descriptor	no	
264	renameat	rename a file relative to directory file descriptors	no	
265	linkat	create a file link relative to directory file descriptors	no	
266	symlinkat	create a symbolic link relative to a directory file descriptor	no	
267	readlinkat	read value of a symbolic link relative to a directory file descriptor	no	
268	fchmodat	change permissions of a file relative to a directory file descriptor	no	
269	faccessat	check user's permissions of a file relative to a directory file descriptor	no	
270	pselect6	synchronous I/O multiplexing	no	
271	ppoll	wait for some event on a file descriptor	no	
272	unshare	disassociate parts of the process execution context	no	
273	set_robust_list	get/set the list of robust futexes	no	
274	get_robust_list	—	—	
275	splice	splice data to/from a pipe	yes[22]	
276	tee	duplicating pipe content	yes[22]	
277	sync_file_range	sync a file segment with disk	no	
278	vmsplice	splice user pages into a pipe	yes[22]	

Number	Name	Description	Flow	Tracked
279	move_pages	move individual pages of a process to another node	no[5]	
280	utimensat	change file timestamps with nanosecond precision	no	
281	epoll_pwait	wait for an I/O event on an epoll file descriptor	no	
282	signalfd	create a file descriptor for accepting signals		
283	timerfd_create	timers that notify via file descriptors	no	
284	eventfd	create a file descriptor for event notification	yes[23]	
285	fallocate	manipulate file space	no	
286	timerfd_settime	timers that notify via file descriptors	yes[24]	
287	timerfd_gettime	—	—	
288	accept4	accept a connection on a socket	no	
289	signalfd4	create a file descriptor for accepting signals	no	
290	eventfd2	create a file descriptor for event notification	no	
291	epoll_create1	open an epoll file descriptor	no	
292	dup3	duplicate a file descriptor	no	
293	pipe2	create pipe	no	
294	inotify_init1	initialize an inotify instance	no	
295	preadv	read or write data into multiple buffers	yes[1]	✓
296	pwritev	—	—	
297	rt_tgsigqueueinfo	queue a signal and data	yes[14]	
298	perf_event_open	set up performance monitoring	no	
299	recvmmsg	receive a message from a socket	yes[8]	✓
300	fanotify_init	initializes the fanotify subsystem	no	
301	fanotify_mark	Management of notification events	no	
302	prlimit64	get and set a process resource limits	no	
303	name_to_handle_at	convert name to handle	no	
304	open_by_handle_at	Open the file handle	no	
305	clock_adjtime	posix clock operation	no	
306	syncfs	commit buffer cache to disk	no	
307	sendmmsg	send a message on a socket	yes[8]	✓
308	setns	reassociate thread with a namespace	no	
309	getcpu	determine CPU and NUMA node on which the calling thread is running	no	

Number	Name	Description	Flow	Tracked
310	process_vm_readv	transfer data between process address spaces	yes[25]	
311	process_vm_writev	—	—	

A.1 Special cases

1. `read, write, open, pread64, pwrite64, readv, writev, sendfile, pipe, dup, dup2, readahead`: we do not directly track all these calls, but instead, we track calls to `read, write`, from/towards the underlying file descriptor or inode, were actual information flows occur.

2. `mprotect`: even though `mprotect` does not directly cause information flows, it changes the protection mode of memory pages. In cases where shared memory mappings exist with other processes (attached via `mmap`[1] with the `MAP_SHARED` flag), it may affect the way information flows occur. As stated in the manpage of `mprotect`: *"On Linux it is always permissible to call mprotect() on any address in a process's address space (except for the kernel vsyscall area). In particular it can be used to change existing code mappings to be writable".* For this reason, we need to hook calls to `mprotect` as well.

3. `munmap`: as stated in the system calls table, this system call does not cause any information flow, however it helps us refine our analysis. When a process shares a memory mapping with another process, there is no way to know which information is swapped between the two, therefore we compute an overestimate of the possible information flows: all information read by one process having *write* access to the memory region is assumed to be read by the other processes having *read* access to it. A call to `munmap` tells us when to stop tracking the caller process (`w.r.t` a given memory mapping). Tracking `munmap` is done by a custom added hook, it is not part of LSM.

4. `rt_ioctl`: this system call manipulates the underlying device parameters of special files. This is commonly used in drivers, for instance, and information may usually be transfered towards a particular device. The last argument of this system call is an untyped pointer to memory, and in some situations, this may possibly lead to information flows between objects of the operating system that we track. However, this case is not handled in our implementation at the moment due to the underlying complexity of hardware drivers. We think reasonable to consider such a case as a hidden channel.

5. `mremap, move_pages`: the pages remain accessible by the same process through its own address space, therefore there is no communication with other processes.

6. `msync`: an information flow occurs, as the corresponding memory mapping is synchronized with its underlying file. However, we track information flows at the level of `shmat` and `shmdt`, and we consider that the mappings are always synchronized (this is an overestimate).

7. `shmat,shmget`: we do not directly track the creation of memory segments by processes with `shmget`, but rather when processes actually attach or detach them to and from their address space, with `shmat` and `shmdt`.

8. `socket, connect, accept, sendto, recvfrom, sendmsg, recvmsg, bind, listen, socketpair`: we do not directly track all these calls, but instead, we track calls to `sendmsg` and `recvmsg`, were actual information flows occur.

9. `clone, fork`:

[1] POSIX says that the behavior of mprotect() is unspecified if it is applied to a region of memory that was not obtained via mmap(2)

- `clone` is mostly used to create threads within one process's address space. If called with `CLONE_VM` or `CLONE_THREAD` flags, the memory space of the parent is shared with the child.
- `fork` is a glibc wrapper, it invokes `clone` with the corresponding flags.

10. `execve`: execute a program. This is tracked in our implementation.

11. `shmdt`: this system call does not cause any information flow. However, as with `munmap`, we need to keep tracks of processes detaching memory segments, in order to stop tracking information flows from and towards to the detached memory segment.

12. `msgsnd, msgrcv, mq_timedsend, mq_timedreceive`: send/receive a message from message queue. This is tracked by our implementation.

13. `ptrace`: information flows are involved when a process is traced: the caller may access information from the child, and communicate information towards the child. Tracing processes as well as accessing sensitive information in `/proc` is tracked by LSM (hooks `security_ptrace_access_check` and `security_ptrace_traceme`). We do not track calls to `ptrace` in our current implementation.

14. `rt_sigqueueinfo`: this system call provides the low-level interface to send a signal plus data to a process or thread. We consider it as a hidden channel, as the main purpose of this interface is signal handling. The receiver of the signal can obtain the accompanying data by establishing a signal handler with the `sigaction(2) SA_SIGINFO` flag. We do not track this in our current implementation. This could be tracked by adding a hook on calls to `sigaction`.

15. `mknod`: file is created empty, therefore there is no information flow.

16. `uselib`: we do not directly track these calls, but we track the underlying calls to `mmap` when mapping the shared library into memory.

17. `swapon`: starts swapping to file/device. Even though swapping involve information flows, we do not track access to the swap area, as is impractical to do so (because swapping is managed by the kernel, and we do not hook kernel code, that we consider as trusted). Accessing the swap area from userspace is only allowed to the system administrator. Future versions of our implementation may restrict access to the swap area from userspace (even to the system administrator).

18. `init_module`: loads an ELF binary into kernel space. This system call requires privileges, and is not tracked by our implementation as it modifies the kernel.

19. `setxattr, lsetxattr, fsetxattr, getxattr, lgetxattr, fsetxattr`: get/set file extended attributes. An information flow occurs and may be used to exchange information between userspace processes. We do not track it in our current implementation, however this is achievable by using LSM hooks (`security_inode_setxattr` and `security_inode_getxattr`). It will be implemented in future releases.

20. `kexec_load`: this is used to load a new kernel at runtime (live booting of a new kernel over the currently running kernel). We do not track such a low-level mechanism: it would be required to flag portions of the memory that are not overwritten by the new kernel.

21. `add_key, request_key`: access kernel's key management facility. This is used *e.g.* to mount remote filesystem which require authentication or a key to enable access. It is possible to use it in a diverted way to establish communication between userspace processes. It is untracked in our current implementation.

22. `splice, tee, vmsplice`: move data between file descriptors without copying between userspace and kernelspace – copy standard output to files and standard output – move user pages into a pipe. Information flows occur between userspace and kernelspace.

23. `eventfd`: can be used by userspace applications as a wait/notify mechanism. Possible hidden channels may be implemented with it. Untracked in our implementation.

24. `timerfs_settime, timerfd_gettime`: those operate on a timer delivering notifications via a file descriptor. These may be used as hidden channels. Untracked in our implementation.

25. `process_vm_readv, process_vm_writev`: transfer data between the address space of two processes (a local process and a remote process). The data is moved directly, without passing through kernel space.